Prepared By
CHETAN D S
Asst. Prof. CSE
RNSIT

UNIT 3: ASSEMBLERS-2

Assembler is a system software which translates assembly language program into Machine Language Program.

3.1 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 sections

3.1.1 Literals:

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

Example:

```
001A        ENDFIL      LDA   =C'EOF'     032010
                        …
                        …
                        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'.

```
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

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.

	START	0
INLOOP	TD	=X’F1’
	JEQ	INLOOP
	RD	=X’F1’
	STCH	DATA
	LTORG	
OUTLP	TD	=X’05’
	JEQ	OUTLP
	LDCH	DATA
	WD	=X’05’
DATA	RESB	1
	END	

LOCCTR				**OBJECT CODE**
		START	0	
0000	INLOOP	TD	=X’F1’	
0003		JEQ	INLOOP	
0006		RD	=X’F1’	
0009		STCH	DATA	
000C		=X’F1’		F1
000D	OUTLP	TD	=X’05’	
0010		JEQ	OUTLP	
0013		LDCH	DATA	
0016		WD	=X’05’	
0019	DATA	RESB	1	
		END		
001A		=X’05		05
001B				

LITTAB

Literal Name	**Value**	**Length**	**Address**
X’F1’	F1	1	000C
X’05’	05	1	001A

3.1.2 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). The general form of the statement is

Symbol EQU value

This statement defines the given symbol (i.e., entering in the SYMTAB) and assigning to it the value specified. The value can be a constant or an expression involving constants and any other symbol which is already defined. One common usage is to define symbolic names that can be used to improve readability in place of numeric values.

For example

```
                    +LDT        #4096
```

This loads the register T with immediate value 4096, this does not clearly what exactly this value indicates. If a statement is included as:

```
          MAXLEN    EQU         4096          and then
                    +LDT        #MAXLEN
```

Then it clearly indicates that the value of MAXLEN is some maximum length value. When the assembler encounters EQU statement, it enters the symbol MAXLEN along with its value in the symbol table. During LDT the assembler searches the SYMTAB for its entry and its equivalent value as the operand in the instruction. The object code generated is the same for both the options discussed, but is easier to understand. If the maximum length is changed from 4096 to 1024, it is difficult to change if it is mentioned as an immediate value wherever required in the instructions. We have to scan the whole program and make changes wherever 4096 is used. If we mention this value in the instruction through the symbol defined by EQU, we may not have to search the whole program but change only the value of MAXLENGTH in the EQU statement (only once).

ORG Statement:

This directive can be used to indirectly assign values to the symbols. The directive is usually called ORG (for origin). Its general format is:

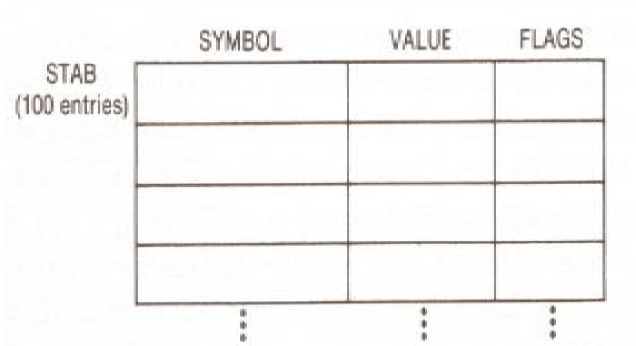
```
ORG         value
```

Where value is a constant or an expression involving constants and previously defined symbols. When this statement is encountered during assembly of a program, the assembler resets its location counter (LOCCTR) to the specified value. Since the values of symbols used as labels are taken from LOCCTR, the ORG statement will affect the values of all labels defined until the next ORG is encountered. ORG is used to control assignment storage in the object program. Sometimes altering the values may result in incorrect assembly.

ORG can be useful in label definition. Suppose we need to define a symbol table with the following structure:

SYMBOL	6 Bytes
VALUE	3 Bytes
FLAG	2 Bytes

The table looks like the one given below.



The symbol field contains a 6-byte user-defined symbol; VALUE is a one-word representation of the value assigned to the symbol; FLAG is a 2-byte field specifies symbol type and other information. The space for the ttable can be reserved by the statement:

```
STAB        RESB        1100
```

If we want to refer to the entries of the table using indexed addressing, place the offset value of the desired entry from the beginning of the table in the index register. To refer to the fields SYMBOL, VALUE, and FLAGS individually, we need to assign the values first as shown below:

```
SYMBOL      EQU         STAB
VALUE       EQU         STAB+6
FLAGS       EQU         STAB+9
```

To retrieve the VALUE field from the table indicated by register X, we can write a statement:

```
LDA     VALUE, X
```

The same thing can also be done using ORG statement in the following way:

```
STAB     RESB    1100
         ORG     STAB
SYMBOL   RESB    6
VALUE    RESW    1
FLAG     RESB    2
         ORG     STAB+1100
```

The first statement allocates 1100 bytes of memory assigned to label STAB. In the second statement the ORG statement initializes the location counter to the value of STAB. Now the LOCCTR points to STAB. The next three lines assign appropriate memory storage to each of SYMBOL, VALUE and FLAG symbols. The last ORG statement reinitializes the LOCCTR to a new value after skipping the required number of memory for the table STAB (i.e., STAB+1100).

Restrictions-EQU

In the case of EQU all the symbols used on the right hand side of the statement must have been defined previously in the program.

ALPHA RESW 1 BETA EQU ALPHA	✓	BETA EQU ALPHA ALPHA RESW 1	✗

Restriction –ORG

All symbols used to specify the new LOCCTR value must have been previously defined.

ALPHA RESB 1 ORG ALPHA BYTE1 RESB 1 BYTE2 RESB 1 BYTE3 RESB 1 ORG	✓	ORG ALPHA BYTE1 RESB 1 BYTE2 RESB 1 BYTE3 RESB 1 ORG ALPHA RESB 1	✗

3.1.3 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. Thus the statement

BUFFEND EQU *

Assigns a value to BUFFEND, which is the address of the next byte following the buffer area. Some values in the object program are relative to the beginning of the program and some are absolute (independent of the program location, like constants). Hence, expressions are classified as either absolute expression or relative expressions depending on the type of value they produce.

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

In the above instruction the difference in the expression gives a value that does not depend on the location of the program and hence gives an absolute immaterial of the relocation of the program. The expression can have only absolute terms. Example:

MAXLEN EQU 1000

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)

3.1.4 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

LOCCTR				**OBJECT CODE**
	READ	START	0	
0000		LDX	#0	
0003		LDT	#11	
0006	MOVECH	JSUB	RDDATA	4B200B
0009		LDCH	DATA	532019
000C		STCH	STR,X	
000F		TIXR	T	
0011		JLT	MOVECH	
		USE	CDATA	
0000	DATA	RESB	1	
0001	STR	RESB	11	
		USE	CBLKS	
0000	BUFFER	RESB	4096	
1000	BUFEND	EQU	*	
1000	MAXLEN	EQU	BUFEND-BUFFER	
		USE		
0014	RDDATA	CLEAR	A	
0016	INLOOP	TD	INDEV	E32018
0019		JEQ	INLOOP	
001C		RD	INDEV	
001F		STCH	DATA	
0022		RSUB		
		USE	CDATA	
000C	INDEV	BYTE	X’F1’	
		END		

BLOCK TABLE

BLOCK NAME	BLOCK NUMBER	ADDRESS	LENGTH
DEFAULT	0	0000	00025
CDATA	1	0025 (0000+0025)	000D
CBLKS	2	0032 (0025+000D)	1000

Program Length = 1032 (0032+1000)

JSUB RDDATA

Opcode	N	I	X	B	P	E	DISPLACEMENT
0100 10	1	1	0	0	1	0	0000 0000 1011

4 B 2 0 0 B

TA of RDDATA = (0000+0014) = 0014 Disp=TA – (PC) = 0014 – 0009 = 00B

LDCH DATA

Opcode	N	I	X	B	P	E	DISPLACEMENT
0101 00	1	1	0	0	1	0	0000 0001 1001

5 3 2 0 1 9

TA of DATA=(0025+0) = 025

Disp = TA – (PC)

= 025 – 00C = 019

TD INDEV

Opcode	N	I	X	B	P	E	DISPLACEMENT
1110 00	1	1	0	0	1	0	0000 0001 1000

E 3 2 0 1 8

TA of INDEV = 000C + 0025 = 031

Disp= TA –(PC) = 031 – 019 = 018

Advantages of Program Blocks

1. We can avoid using Format 4
2. Base register is no longer necessary.
3. The problem of placement of literals is easily solved.
4. Program readability is improved.

3.1.5 CONTROL SECTIONS and PROGRAM LINKING

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 of a program.

The syntax

secname CSECT

– separate location counter for each control section

LOCCTR	SOURCE STATEMENT			OBJECT CODE
	READ	START	0	
		EXTDEF	DATA	
		EXTREF	RDDATA	
0000		LDX	#0	
0003		LDT	#11	
0006	MOVECH	+JSUB	RDDATA	4B100000
000A		LDCH	DATA	
000D		STCH	STR,X	
000F		TIXR	T	
0011		JLT	MOVECH	
0014	DATA	RESB	1	
0015	STR	RESB	11	
0020				
	RDDATA	CSECT		
		EXTREF	DATA	
0000		CLEAR	A	B400
0002	INLOOP	TD	INDEV	E3200D
0005		JEQ	INLOOP	332FFA
0008		RD	INDEV	DB2007
000B		+STCH	DATA	57100000
000F		RSUB		4F0000
0012	INDEV	BYTE	X’F1’	F1
0013		END		

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

EXTREF (external Reference):

It names symbols that are used in this CONTROL section and are defined elsewhere.

For Program Linking we require Define, Refer and Modification Record.

1. Define Record: Lists symbols that are defined in this control section.

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

2. Refer Record

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

3. Modification Record

Col. 1	M
Col. 2-7	Starting address of the field to be modified (hexadecimal), relative to the beginning of control section.
Col. 8-9	Length of the field to be modified, in half-bytes (hexadecimal)
Col. 10	Modification flag(+ or -)
Col.11-16	External symbol whose value is to be added to or subtracted from the indicated field

Handling External Reference

MOVECH +JSUB RDDATA 4B100000

The operand RDDATA is an external reference.

- The assembler has no idea where RDDATA is
- inserts an address of zero
- can only use extended formatto provide enough room (that is, relative addressing for external reference is invalid)

The assembler generates information for each external reference that will allow the loader to perform the required linking.

Similarly for the instruction **+STCH DATA** the object code is 5 7 100000

HRDDATA000000000013

RDATA

T00000013B400E3200D332FFADB2007571000004F0000F1

M00000C05+DATA

E

Figure: Object Program for control section-**RDDATA**

3.2 ASSEMBLER DESIGN OPTIONS

3.2.1 ONE PASS ASSEMBLERS

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)

There are two types of one-pass assemblers:

One that produces object code directly in memory for immediate execution (Load-and-go assemblers).

The other type produces the usual kind of object code for later execution.

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.

LOCCTR				**OBJECT CODE**
	READ	START	1000	
1000	ZERO	WORD	0	000000
1003	ELEVEN	WORD	11	00000B
1006	DATA	RESB	1	
1007	STR	RESB	11	
1012		LDX	ZERO	041000
1015	MOVECH	JSUB	RDDATA	48**0000**
1018		LDCH	DATA	501006
101B		STCH	STR,X	549007
101E		TIX	ELEVEN	2C100C
1021		JLT	MOVECH	381015
1024	INDEV	BYTE	X’F1’	F1
1025	RDDATA	LDA	ZERO	001000
1028	INLOOP	TD	INDEV	
102B		JEQ	INLOOP	
102E		RD	INDEV	
1031		STCH	DATA	
1034		RSUB		
1037		END		

OPTAB	
MNEMONIC	**OPCODE**
LDX	04
LDCH	50
STCH	54
TIX	2C
JLT	38
LDA	00
JSUB	48
TD	E0
JEQ	30
RD	D8
RSUB	4C

SYMTAB

LABEL	**ADDRESS**			
ZERO	1000			
ELEVEN	1003			
DATA	1006			
STR	1007			
MOVECH	1015			
RDDATA	*	→	1016	NULL

* indicate undefined symbol

Figure: The status after scanning the input statement **MOVECH JSUB RDDATA**

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.

HREAD 001000

T00100006000000000000B

T0010121604100048**00000**5010065490072C100C381015F1001000

T001016021025

Figure: Object Program after scanning line **RDDATA LDA ZERO.**

Example 2:

LOCCTR	**SOURCE STATEMENT**			**Object Code**
	STORE	START	0	
0000		LDA	TEN	000000
0003		STA	XYZ	0C0000
0006	TEN	WORD	10	00000A
0009	XYZ	RESW	1	
000C		END		

OPTAB	
MNEMONIC	**OPCODE**
LDA	00
STA	0C

SYMTAB				
LABEL	**ADDRESS**			
TEN	*	→	0001	NULL
XYZ	*	→	0004	NULL

Figure: Symbol Table Entries for the above program after scanning the input **STA XYZ**.

SYMTAB

LABEL	**ADDRESS**
TEN	0006
XYZ	0009

Figure: Symbol Table Entries for the above program after scanning the input **XYZ RESW 1.**

HSTORE 00000000000C

T000000090000000C000000000A

T000001020006

T000004020009

E000000

Figure: Object Program

3.2.2 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
```

- 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.

Implementation Issues for Modified Two-Pass Assembler:

Implementation Isuues when forward referencing is encountered in *Symbol Defining statements* :

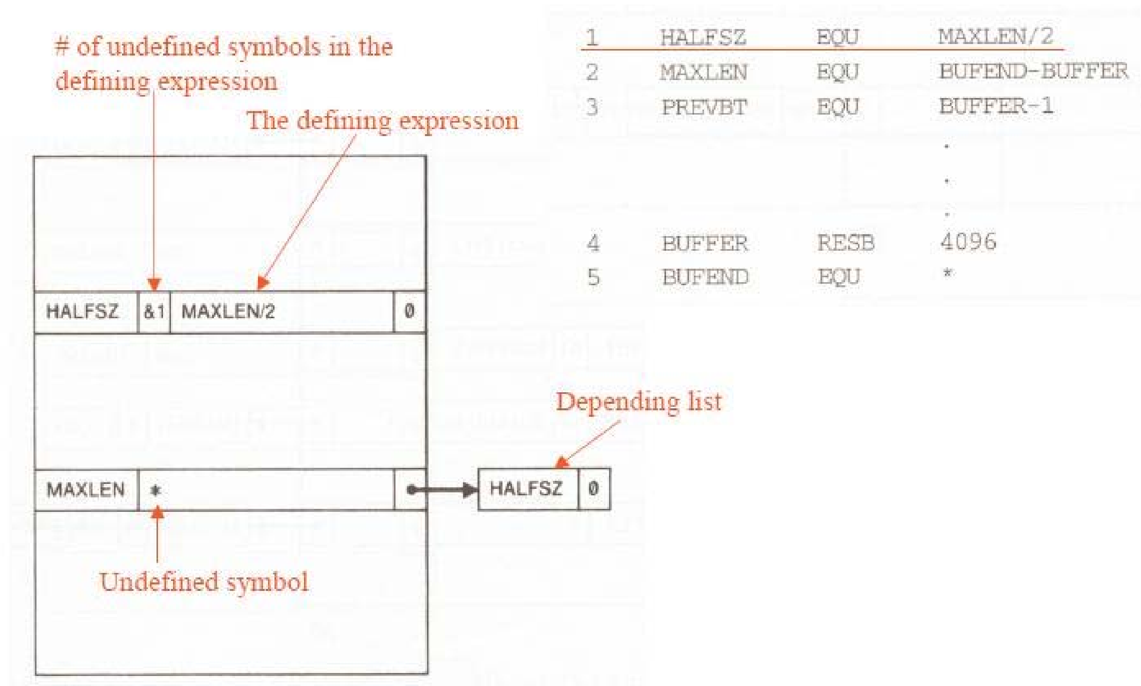
For a forward reference in symbol definition, we store in the SYMTAB:

- The symbol name
- The defining expression
- The number of undefined symbols in the defining expression

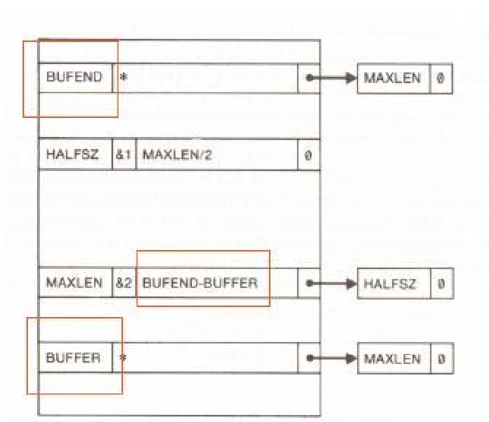
The undefined symbol (marked with a flag *) associated with a list of symbols depend on this undefined symbol.

When a symbol is defined, we can recursively evaluate the symbol expressions depending on the newly defined symbol.

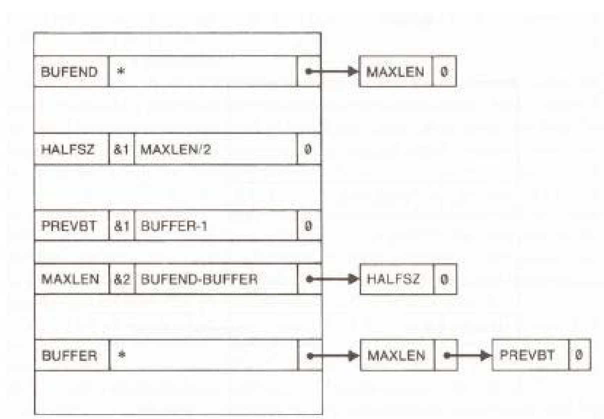
Multi-Pass Assembler Example Program



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

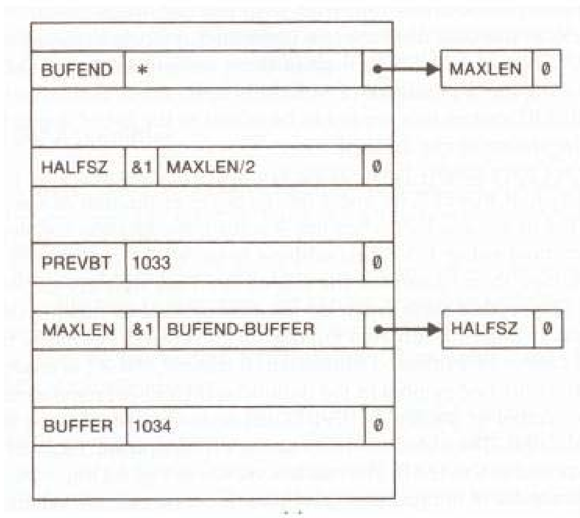


2 MAXLEN EQU BUFEND-BUFFER

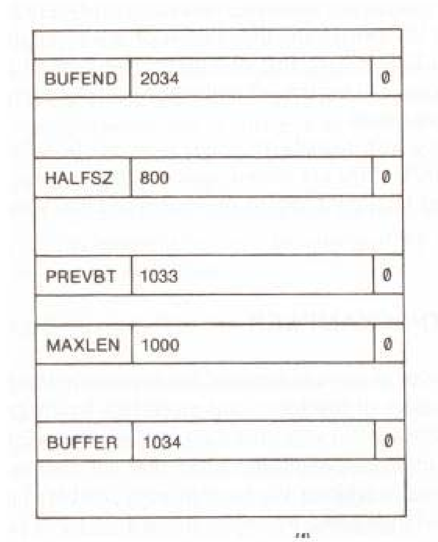


3 PREVBT EQU BUFFER-1

Let us assume that when line 4 is read, the location counter contains the hexadecimal value 1034.



4 BUFFER RESB 4096



5 BUFEND EQU *

In Multi-Pass Assembler the portion of the program that involve forward references in symbol definitions are saved during Pass1. Additional Passes through these stored definitions are made as the assembly progresses. This process is followed by a normal Pass 2.

***