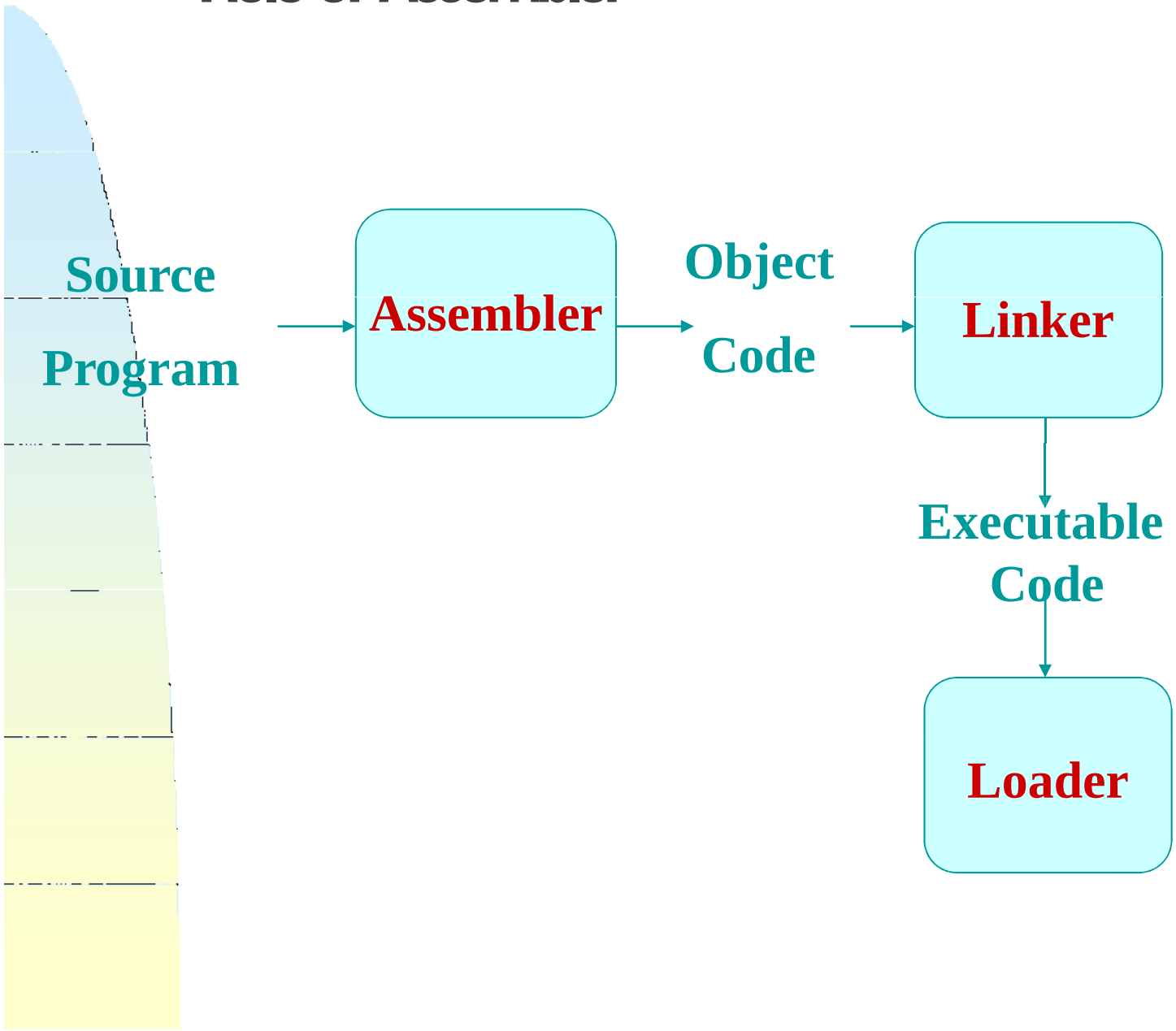
Assemblers

System Software
by Leland L. Beck
Chapter 2

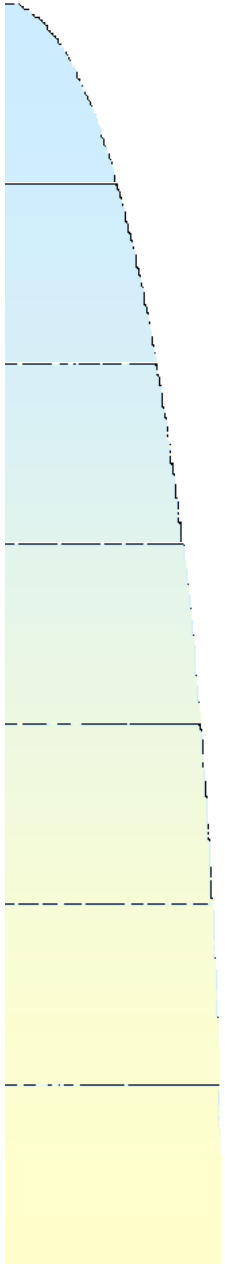
Role of Assembler



Chapter 2 -- Outline

- Basic Assembler Functions
- Machine-dependent Assembler Features
- Machine-independent Assembler Features
- Assembler Design Options

Introduction to Assemblers



- **Fundamental functions**
  - translating mnemonic operation codes to their machine language equivalents
  - assigning machine addresses to symbolic labels
- **Machine dependency**
  - different machine instruction formats and codes

Example Program (Fig. 2.1)

- **Purpose**
  - reads records from input device (code F1)
  - copies them to output device (code 05)
  - at the end of the file, writes EOF on the output device, then RSUB to the operating system

Line	Loc	Source Statement			Object Code
5	1000	COPY	START	1000	-----
10	1000	FIRST	STL	RETADR	141033
15	1003	CLOOP	JSUB	RDREC	482039
20	1006		LDA	LENGTH	00 1036
25	1009		COMP	ZERO	281030
30	100C		JEQ	ENDFIL	301015
35	100F		JSUB	WRREC	482061
40	1012		J	CLOOP	3C1003
45	1015	ENDFIL	LDA	EOF	00102A
50	1018		STA	BUFFER	0C1039
55	101B		LDA	THREE	00102D
60	101E		STA	LENGTH	0C1036
65	1021		JSUB	WRREC	482061
70	1024		LDL	RETADR	0 81033
75	1027		RSUB		4C0000
80	102A	EOF	BYTE	C'EOF'	454F46
85	102D	THREE	WORD	3	0 00003
90	1030	ZERO	WORD	0	0 00000
95	1033	RETADR	RESW	1	----
100	1036	LENGTH	RESW	1	----
105	1039	BUFFER	RESB	4096	-----

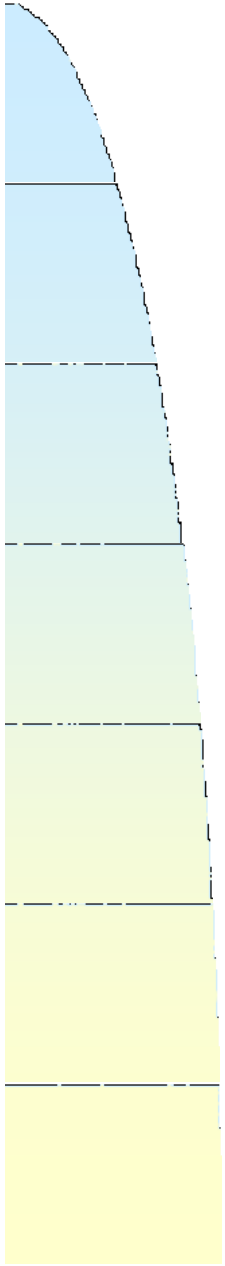
110 115 120		. . Subroutine to read record into BUFFER .			
125	2039	RDREC	LDX	ZERO	0 41030
130	203C		LDA	ZERO	00 1030
135	203F	RLOOP	TD	INPUT	E0205D
140	2042		JEQ	RLOOP	30203F
145	2045		RD	INPUT	D8205D
150	2048		COMP	ZERO	281030
155	204B		JEQ	EXIT	302057
160	204E		STCH	BUFFER,X	549039
165	2051		TIX	MAXLEN	2C205E
170	2054		JLT	RLOOP	38203F
175	2057	EXIT	STX	LENGTH	101036
180	205A		RSUB		4C0000
185	205D	INPUT	BYTE	X'F1'	F1
190	205E	MAXLEN	WORD	4096	00 1000

195 200 205		. . Subroutine to write record from BUFFER .			
210	2061	WRREC	LDX	ZERO	0 41030
215	2064	WLOOP	TD	OUTPUT	E02079
220	2067		JEQ	WLOOP	302064
225	206A		LDCH	BUFFER,X	509039
230	206D		WD	OUTPUT	DC2079
235	2070		TIX	LENGTH	2C1036
240	2073		JLT	WLOOP	382064
245	2076		RSUB		4C0000
250	2079	OUTPUT	BYTE	X'05'	0 5
255			END	FIRST	

Example Program (Fig. 2.1)

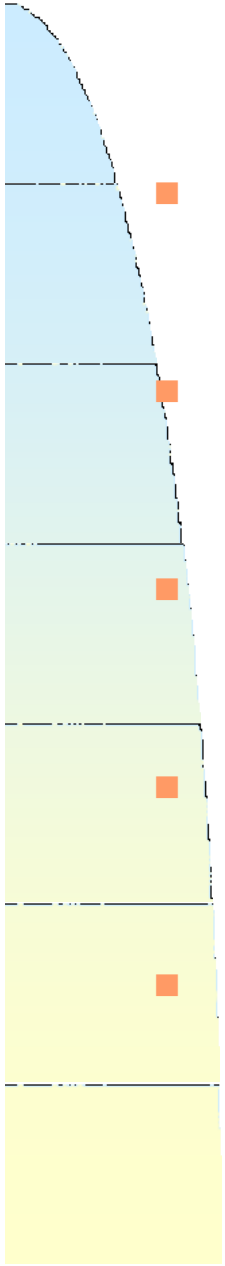
- **Data transfer (RD, WD)**
  - a buffer is used to store record
  - buffering is necessary for different I/O rates
  - the end of each record is marked with a null character (00_{16})
  - the end of the file is indicated by a zero-length record
- **Subroutines (JSUB, RSUB)**
  - RDREC, WRREC
  - save link register first before nested jump

Assembler Directives



- **Pseudo-Instructions**
  - Not translated into machine instructions
  - Providing information to the assembler
- **Basic assembler directives**
  - START
  - END
  - BYTE
  - WORD
  - RESB
  - RESW

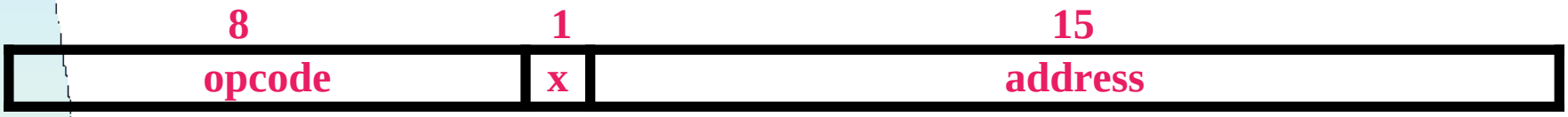
Assembler's functions



- Convert mnemonic operation codes to their machine language equivalents
- Convert symbolic operands to their equivalent machine addresses ✓
- Build the machine instructions in the proper format
- Convert the data constants to internal machine representations
- Write the object program and the assembly listing

Example of Instruction Assemble

STCH BUFFER,X 549039

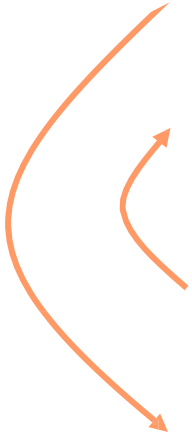


$(54)_{16}$ 1 $(001)_2$ $(039)_{16}$

- Forward reference

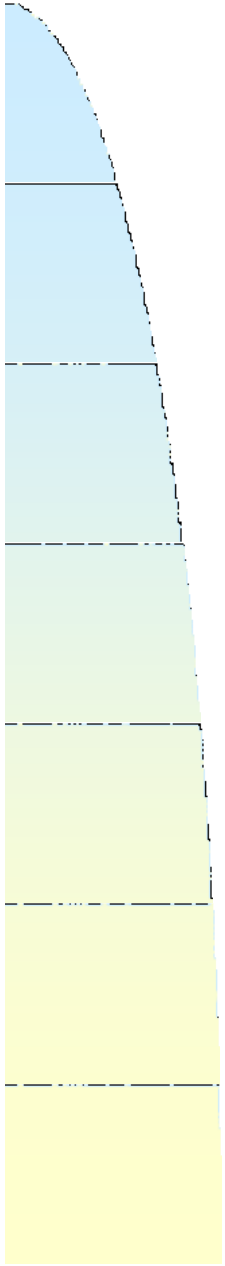
Difficulties: Forward Reference

- **Forward reference: reference to a label that is defined later in the program.**



Loc	Label	Operator	Operand	
1000	FIRST	STL	RETADR	
1003	CLOOP	JSUB	RDREC	
…	…	…	…	…
1012		J	CLOOP	
…	…	…	…	…
1033	RETADR	RESW	1	

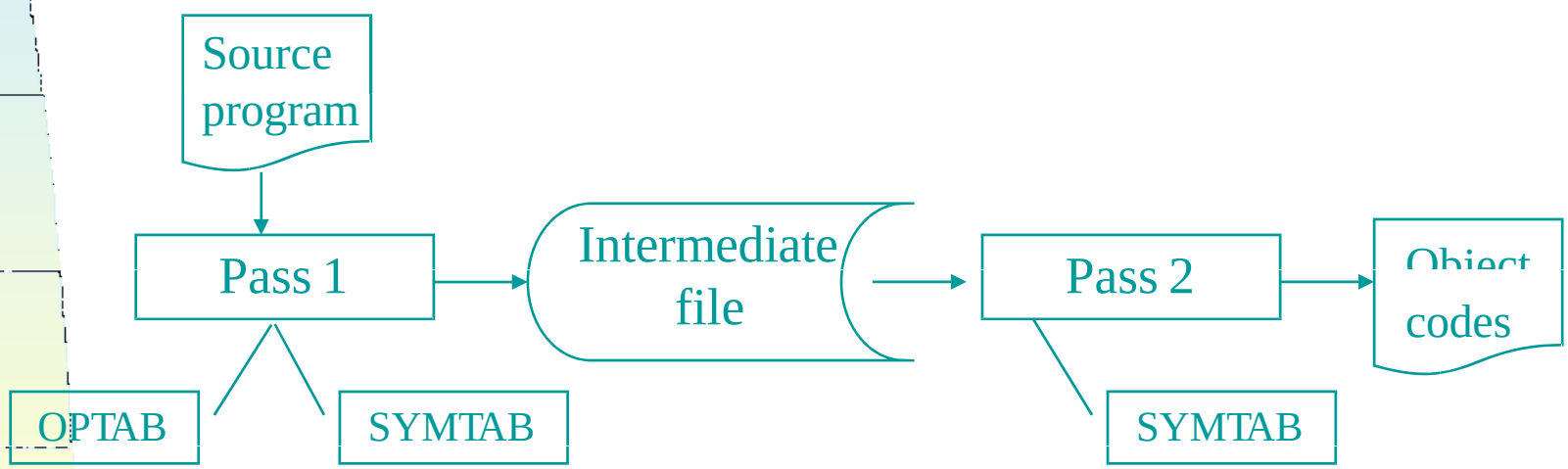
Two Pass Assembler



- **Pass 1**
  - Assign addresses to all statements in the program
  - Save the values assigned to all labels for use in Pass 2
  - Perform some processing of assembler directives
- **Pass 2**
  - Assemble instructions
  - Generate data values defined by BYTE, WORD
  - Perform processing of assembler directives not done in Pass 1
  - Write the object program and the assembly listing

Two Pass Assembler

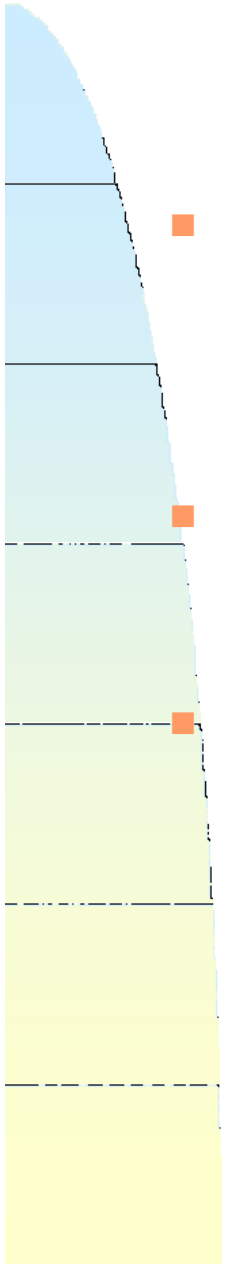
- **Read from input line**
  - LABEL, OPCODE, OPERAND



Data Structures

- Operation Code Table (OPTAB)
- Symbol Table (SYMTAB)
- Location Counter(LOCCTR)

OPTAB (operation code table)



- **Content**
  - menmonic, machine code (instruction format, length) etc.
- **Characteristic**
  - static table
- **Implementation**
  - array or hash table, easy for search

SYMTAB (symbol table)

- **Content**
  - label name, value, flag, (type
- **Characteristic**
  - dynamic table (insert, delete
- **Implementation**
  - hash table, non-random key

COPY	1000
FIRST	1000
CLOOP	1003
ENDFIL	1015
EOF	1024
THREE	102D
ZERO	1030
RETADR	1033
LENGTH	1036
BUFFER	1039
RDREC	2039

n

Object Program

- **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)

Fig. 23

```
H COPY  001000 00107A
T 001000 1E 141033 482039 001036 281030 301015 482061 ...
T 00101E 15 0C1036 482061 081044 4C0000 454F46 000003 000000
T 002039 1E 041030 001030 E0205D 30203F D8205D 281030 …
T 002057 1C 101036 4C0000 F1 001000 041030 E02079 302064 …
T 002073 07 382064 4C0000 05
E 001000
```

Homework #1

```
SUM       START   4000
FIRST     LDX     ZERO
          LDA     ZERO
LOOP      ADD     TABLE,X
          TIX     COUNT
          JLT     LOOP
          STA     TOTAL
          RSUB
TABLE     RESW    2000
COUNT     RESW    1
ZERO      WORD    0
TOTAL     RESW    1
          END     FIRST
```

End of Sec 2-1

Assembler Design

- **Machine Dependent Assembler Features**
  - instruction formats and addressing modes
  - program relocation
- **Machine Independent Assembler Features**
  - literals
  - symbol-defining statements
  - expressions
  - program blocks
  - control sections and program linking

Algorithm of PASS 1 of TWO-PASS ASSEMBLER :

```
Begin
read first input line
if OPCODE = „START“ then
begin
save #[Operand] as starting address
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
insert (LABEL,LOCCTR) into SYSTAB
end (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}
```

- The algorithm scans the first statement START and saves the operand field (the address) as the starting address of the program. Initializes the LOCCTR value to this address. This line is written to the intermediate line.
- If no operand is mentioned the LOCCTR is initialized to zero. If a label is encountered, the symbol has to be entered in the symbol table along with its associated address value.
- If the symbol already exists that indicates an entry of the same symbol already exists. So an error flag is set indicating a duplication of the symbol.
- It next checks for the mnemonic code, it searches for this code in the OPTAB. If found then the length of the instruction is added to the LOCCTR to make it point to the next instruction.
- If the opcode is the directive WORD it adds a value 3 to the LOCCTR. If it is RESW, it needs to add the number of data word to the LOCCTR. If it is BYTE it adds a value one to the LOCCTR, if RESB it adds number of bytes.
- If it is END directive then it is the end of the program it finds the length of the program by evaluating current LOCCTR – the starting address mentioned in the operand field of the END directive. Each processed line is written to the intermediate file.

Algorithm of PASS 2 of TWO-PASS ASSEMBLER :

```
Begin
read first input line { from intermediate file}
if OPCODE = "START" then
begin
write listing line
read next input line
end {if START}
write Header record to object program.
Initialize first Text record
While OPCODE != "END"                    do
                                         begin
                                         if this is not a comment line then
                                         begin
                                         search OPTAB for OPERAND
                                         if found then
                                         begin
                                         if there is a symbol in OPERAND field then
                                         search SYMTAB for OPERAND
                                         if found then
```

```
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
end {if opcode found}
else if OPCODE = 'BYTE' or 'WORD' then
convert constant to object code
if object code will not fit into the current Text record then
begin
write Text record to object program
initialize new Text record
end
add object code to Text record
end {if not comment}
write listing line
read next input line
end {while not END}
read last Text record to object program
write End record to object program
write last listing line
```

End {pass 2}

- The algorithm scans the first input line from intermediate file & if it is START then as it is written & reads the next input line.
- First line will be written to object program in form of Header record format.
- Initializes the Text record. Until the OPCODE != END all the statements object code will be written to Text Record by searching OPTAB & SYMTAB for operand.
- If it is not possible to fit the object code into the current Text record then it initializes another Text record to which remaining object code will be written.
- After writing the last Text record to the object program, then write the End record to the object program.

Machine-dependent Assembler Features

Sec. 2-2

- Instruction formats and addressing modes
- Program relocation

Instruction Format and Addressing Mode

- **SIC/XE**
  - PC-relative or Base-relative addressing: op m
  - Indirect addressing: op @m
  - Immediate addressing: op #c
  - Extended format: +op m
  - Index addressing: op m,x
  - register-to-register instructions
  - larger memory -> multi-programming (program allocation)
- **Example program**
  - Figure 2.5

Translation

- **Register translation**
  - register name (A, X, L, B, S, T, F, PC, SW) and their values (0,1, 2, 3, 4, 5, 6, 8, 9)
  - preloaded in SYMTAB
- **Address translation**
  - Most register-memory instructions use program counter relative or base relative addressing
  - Format 3: 12-bit address field
    - base-relative: 0~4095
    - pc-relative: -2048~2047
  - Format 4: 20-bit address field

PC-Relative Addressing Modes

- **PC-relative**
  - 10 0000 FIRST STL RETADR 17202D

$(14)_{16}$ 1 1 0 0 1 0 $(02D)_{16}$

☞ displacement= RETADR - PC = 30-3 = 2D

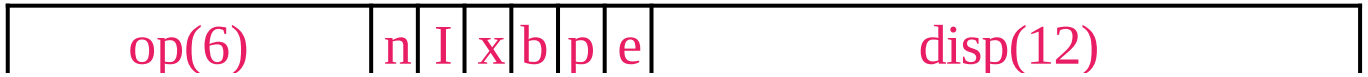
  - 40 0017 J CLOOP 3F2FEC

$(3C)_{16}$ 1 1 0 0 1 0 $(FEC)_{16}$

☞ displacement= CLOOP-PC= 6 - 1A= -14= FEC

Base-Relative Addressing Modes

- **Base-relative**
  - base register is under the control of the programmer
  - 12 LDB #LENGTH
  - 13 BASE LENGTH
  - 160 104F STCH BUFFER, X 57C003



(54)$_{16}$ 1 1 1 1 0 0 (003) $_{16}$

(54) 1 1 1 0 1 0 0036-1051= -101B

  - displacement= BUFFER - B = 0036 - 0033 = 3
  - NOBASE is used to inform the assembler that the contents of the base register no longer be relied upon for addressing

Immediate Address Translation

- **Immediate addressing**
  - 55 0020 LDA #3 010003

(00)$_{16}$ 0 1 0 0 0 0 (003) $_{16}$

133 103C +LDT #4096 75101000

(74)$_{16}$ 0 1 0 0 0 1 (01000) $_{16}$

Immediate Address Translation (Cont.)

- **Immediate addressing**
  - 12 0003 LDB #LENGTH 69202D

(68)$_{16}$ 0 1 0 0 1 0 (02D) $_{16}$

(68)$_{16}$ 0 1 0 0 0 0 (033)$_{16}$ 690033

  - the immediate operand is the symbol LENGTH
  - the address of this symbol LENGTH is loaded into register B
  - LENGTH=0033=PC+displacement=0006+02D
  - if immediate mode is specified, the target address becomes the operand

Indirect Address Translation

- **Indirect addressing**
  - target addressing is computed as usual (PC-relative or BASE-relative)
  - only the n bit is set to 1
  - 70 002A J @RETADR 3E2003

(3C)$_{16}$ 1 0 0 0 1 0 (003)$_{16}$

- TA=RETADR=0030
- TA=(PC)+disp=002D+0003

Program Relocation

- **Example Fig. 2.1**
  - *Absolute program*, starting address 1000

    e.g. 55 101B LDA THREE 00102D
  - Relocate the program to 2000

    e.g. 55 101B LDA THREE 00202D
  - Each Absolute address should be modified
- **Example Fig. 2.5:**
  - Except for absolute address, the rest of the instructions need not be modified
    - not a memory address (immediate addressing)
    - PC-relative, Base-relative
  - The only parts of the program that require modification at load time are those that specify direct addresses

The relocation program can be solved in the following way:

1. When the assembler generates the object code for the JSUB instruction, it will insert the address of RDREC relative to the start of the program.
2. The assembler will also produce a command for the loader, instructing it to add the beginning address of the program to the address field in the JSUB instruction at load time.

The command for the loader must also be a part of the object program. We can accomplish this with a Modification record having the following format :

Modification Record :

Col. 1 : M
Col. 2-7 : Starting location of the address field to be modified, relative to the beginning of the program(hexadecimal).
Col. 8-9 : Length of the address field to be modified, in half-bytes(Hexadecimal).

For JSUB instruction we are using an example, the Modification record would be

M^000007^05

Example

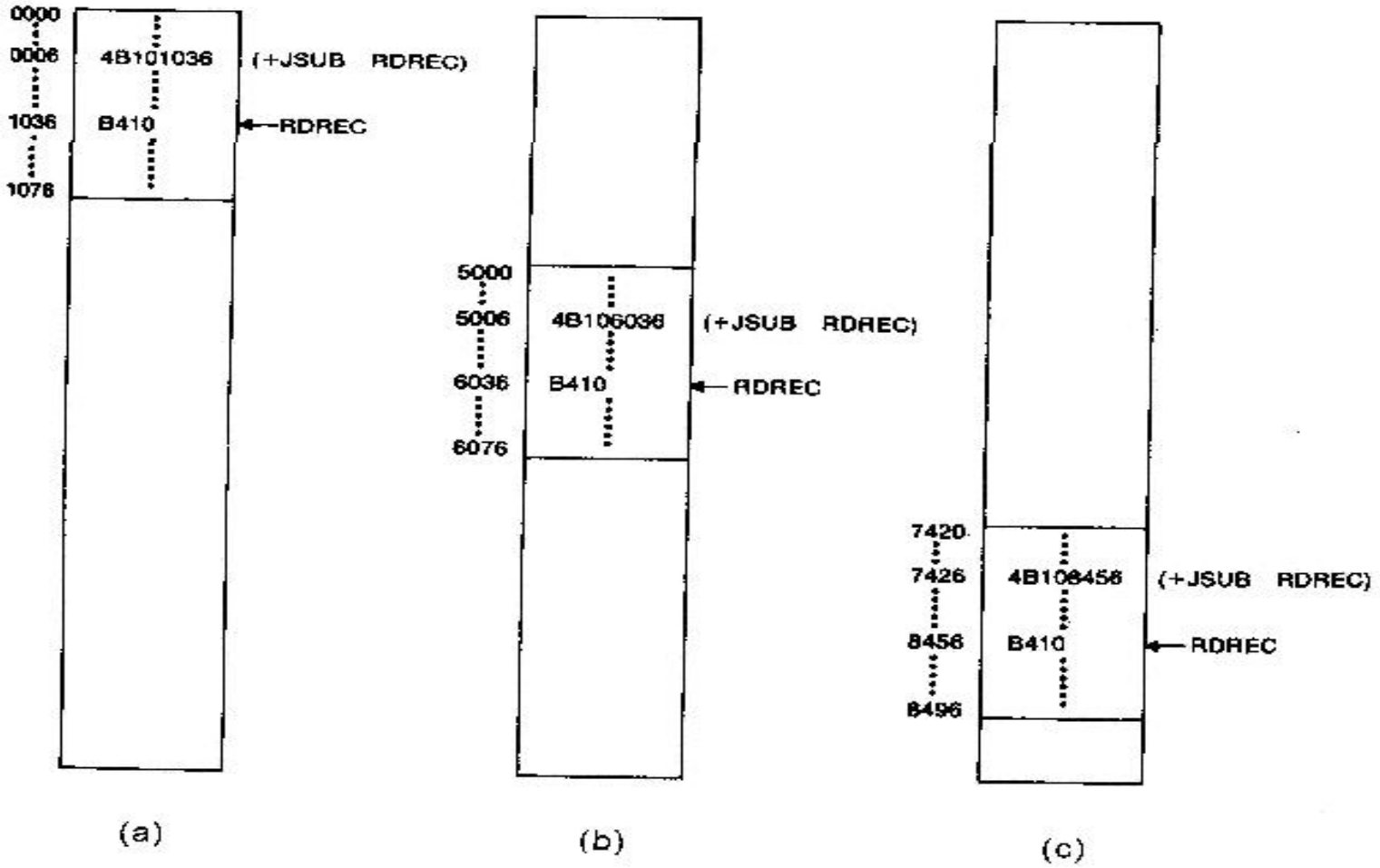
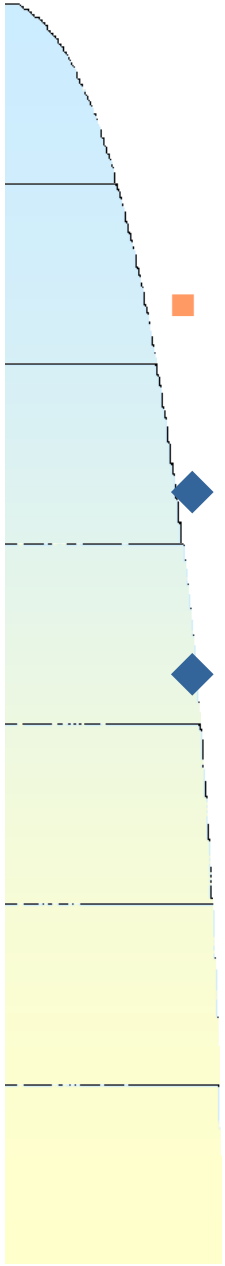


圖2.7 程式重定位範例

Relocatable Program



- **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

Object Code

```
HCOPY  000000001077
T00000001D17202D69202D4B10103603202629000033200074B10105D3F2FEC032010
T00001D130F2016010003 0F200D4B10105D3E2003454F46
T001036 1DB410B400B44075101000E32019332FFADB2013A00433200857C003B850
T00105 31D3B2FEA1340004F0000F1B410774000E32011332FFA53C003DF2008B850
T00107 0073B2FEF4F000005
M00000705
M00001405
M00002705
E000000
```

圖2.8　相對於圖2.6的目的程式

End of Sec 2-2

Machine-Independent Assembler Features

- Literals
- Symbol Defining Statement
- Expressions
- Program Blocks
- Control Sections and Program Linking

Literals

- **Design idea**
  - Let programmers to be able to write the value of a constant operand as a part of the instruction that uses it.
  - This avoids having to define the constant elsewhere in the program and make up a label for it.
- **Example**
  - e.g. 45 001A ENDFILLDA =C’EOF’ 032010
  - 93 LTORG
  - 002D * =C’EOF’ 454F46
  - e.g. 215 1062 WLOOP TD =X’05’ E32011

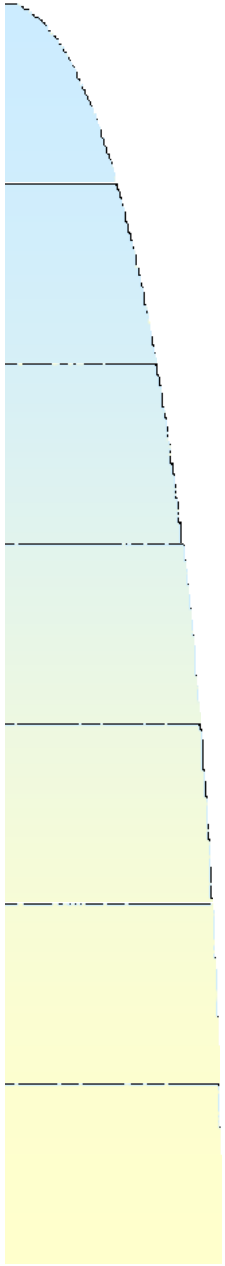
Literals vs. Immediate Operands

- **Immediate Operands**
  - The operand value is assembled as part of the machine instruction
  - e.g. 55 0020 LDA #3 010003
- **Literals**
  - The assembler generates the specified value as a constant at some other memory location
  - e.g. 45 001A ENDFILLDA =C'EOF' 032010
- **Compare (Fig. 2.6)**
  - e.g. 45 001A ENDFIL LDA EOF 032010
  - 80 002D EOF BYTE C'EOF 454F46

Literal - Implementation (1/3)

- **Literal pools**
  - Normally literals are placed into a pool at the end of the program
    - see Fig. 2.10 (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
    - reason: keep the literal operand close to the instruction

Literal - Implementation (2/3)



- **Duplicate literals**
  - e.g. 215 1062 WLOOP TD =X'05'
  - e.g. 230 106B WD =X'05'
  - The assemblers should recognize duplicate literals and store only one copy of the specified data value
    - Comparison of the defining expression
      - Same literal name with different value, e.g. LOCCTR=*
    - Comparison of the generated data value
      - The benefits of using generate data value are usually not great enough to justify the additional complexity in the assembler

Literal - Implementation (3/3)

- **LITTAB**
  - literal name, the operand value and length, the address assigned to the operand
- **Pass 1**
  - build LITTAB with literal name, operand value and length, leaving the address unassigned
  - when LTORG statement is encountered, assign an address to each literal not yet assigned an address
- **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

Symbol-Defining Statements

- **Labels on instructions or data areas**
  - the value of such a label is the address assigned to the statement
- **Defining symbols**
  - symbol EQU value
  - value can be: ✶ constant, ✦ other symbol, ✹ expression
  - making the source program easier to understand
  - no forward reference

Symbol-Defining Statements

- **Example 1**
  - MAXLEN EQU 4096
  - +LDT #MAXLEN

```
        +LDT    #4096
```

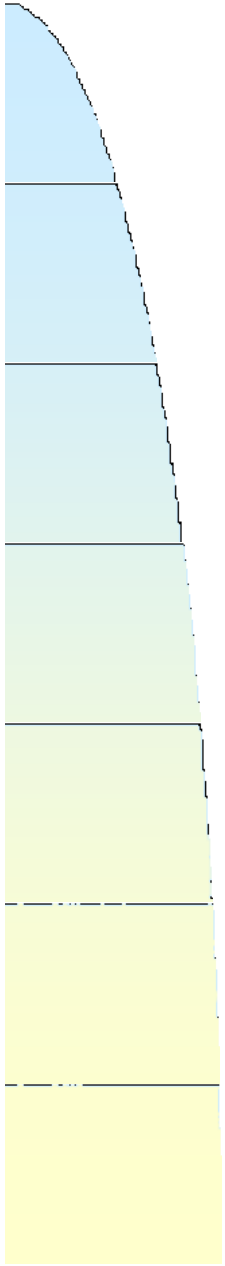
- **Example 2**
  - BASE EQU R1
  - COUNT EQU R2
  - INDEX EQU R3
- **Example 3**
  - MAXLEN EQU BUFFEND-BUFFER

ORG (origin)

- **Indirectly assign values to symbols**
- **Reset the location counter to the specified value**
  - ORG value
- **Value can be: ✶ constant, ✦ other symbol, ✹ expression**
- **No forward reference**
- **Example**
  - SYMBOL: 6bytes
  - VALUE: 1word
  - FLAGS: 2bytes
  - LDA VALUE, X

	SYMBOL	VALUE	FLAGS
STAB			
(100 entries)			
	.	.	.
	.	.	.
	.	.	.

ORG Example



- **Using EQU statements**
  - STAB RESB 1100
  - SYMBOL EQU STAB
  - VALUE EQU STAB+6
  - FLAG EQU STAB+9
- **Using ORG statements**
  - STAB RESB 1100
  - ORG STAB
  - SYMBOL RESB 6
  - VALUE RESW 1
  - FLAGS RESB 2
  - ORG STAB+1100

Expressions

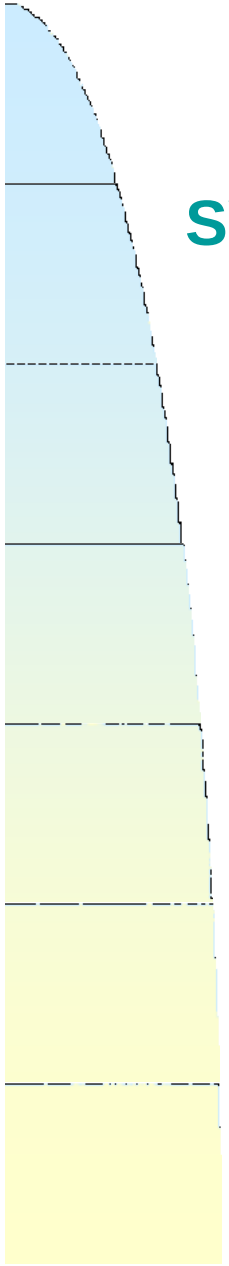
- **Expressions can be classified as absolute expressions or relative expressions**
  - MAXLEN EQU BUFEND-BUFFER
  - BUFEND and BUFFER both are relative terms, representing addresses within the program
  - However the expression BUFEND-BUFFER represents an absolute value
- **When relative terms are paired with opposite signs, the dependency on the program starting address is canceled out; the result is an absolute value**

SYMTAB

- None of the relative terms may enter into a multiplication or division operation
- Errors:
  - BUFEND+BUFFER
  - 100-BUFFER
  - 3*BUFFER
- The type of an expression

Symbol	Type	Value

Example 2.9



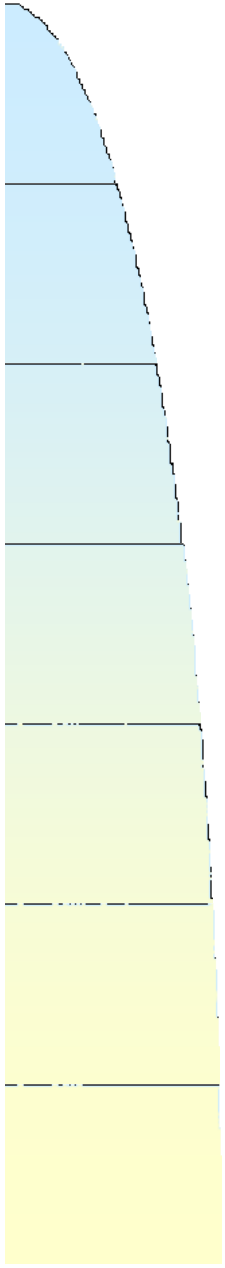
SYMTAB

Name	Value
COPY	0
FIRST	0
CLOOP	6
ENDFIL	1A
RETADR	30
LENGTH	33
BUFFER	36
BUFEND	1036
MAXLEN	1000
RDREC	1036
RLOOP	1040

LITTAB

C'EOF'	454F46	3	002D
X'05'	05	1	1076

Program Blocks



- **Program blocks**
  - refer to segments of code that are rearranged within a single object program unit
  - **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
  - Example: Figure 2.11
  - Each program block may actually contain several separate segments of the source program

Program Blocks - Implementation

- **Pass 1**
  - each program block has a separate location counter
  - each label is assigned an address that is relative to the start of the block that contains it
  - at the end of Pass 1, the latest value of the location counter for each block indicates the length of that block
  - the assembler can then assign to each block a starting address in the object program
- **Pass 2**
  - The address of each symbol can be computed by adding the assigned block starting address and the relative address of the symbol to that block

Figure 2.12

- **Each source line is given a relative address assigned and a block number**

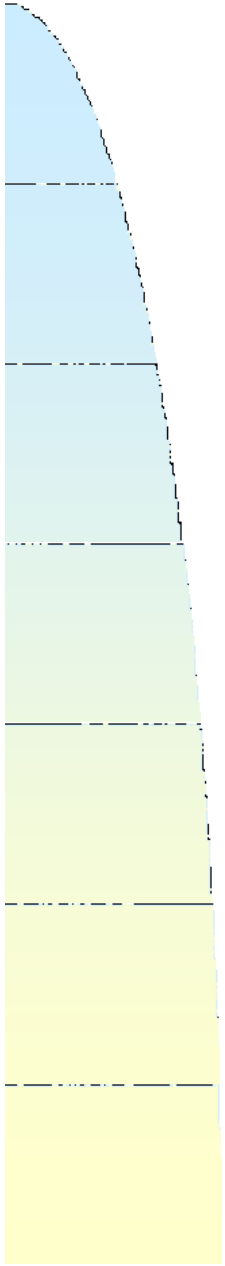
Block name	Block number	Address	Length
(default)	0	0000	0066
CDATA	1	0066	000B
CBLKS	2	0071	1000

- **For absolute symbol, there is no block number**
  - line 107
- **Example**
  - 20 0006 0 LDA LENGTH 032060
  - LENGTH=(Block 1)+0003= 0066+0003= 0069
  - LOCCTR=(Block 0)+0009= 0009

Program Readability

- **Program readability**
  - No extended format instructions on lines 15, 35, 65
  - No needs for base relative addressing (line 13, 14)
  - LTORG is used to make sure the literals are placed ahead of any large data areas (line 253)
- **Object code**
  - It is not necessary to physically rearrange the generated code in the object program
  - see Fig. 2.13, Fig. 2.14

Control Sections and Program Linking



- Control Sections
  - are most often used for subroutines or other logical subdivisions of a program
  - the programmer can assemble, load, and manipulate each of these control sections separately
  - instruction in one control section may need to refer to instructions or data located in another section
  - because of this, there should be some means for linking control sections together
  - Fig. 2.15, 2.16

External Definition and References

- **External definition**
  - **EXTDEF name [, name]**
  - EXTDEF names symbols that are defined in this control section and may be used by other sections
- **External reference**
  - **EXTREF name [,name]**
  - EXTREF names symbols that are used in this control section and are defined elsewhere
- **Example**
  - 15 0003 CLOOP +JSUB RDREC 4B100000
  - 160 0017 +STCH BUFFER,X 57900000
  - 190 0028 MAXLEN WORD BUFEND-BUFFER 000000

Implementation

- **The assembler must include information in the object program that will cause the loader to insert proper values where they are required**
- **Define record**
  - Col. 1 D
  - Col. 2-7 Name of external symbol defined in this control section
  - Col. 8-13 Relative address within this control section (hexadeccimal)
  - Col.14-73 Repeat information in Col. 2-13 for other external symbols
- **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

Modification Record

- **Modification record**
  - Col. 1 M
  - Col. 2-7 Starting address of the field to be modified (hexiadecimal)
  - Col. 8-9 Length of the field to be modified, in half-bytes (hexadeccimal)
  - Col.11-16 External symbol whose value is to be added to or subtracted from the indicated field
  - Note: control section name is automatically an external symbol, i.e. it is available for use in Modification records.
- **Example**
  - Figure 2.17
  - M00000405+RDREC
  - M00000705+COPY

External References in Expression

- **Earlier definitions**
  - required all of the relative terms be paired in an expression (an absolute expression), or that all except one be paired (a relative expression)
- **New restriction**
  - Both terms in each pair must be relative within the same control section
  - Ex: BUFEND-BUFFER
  - Ex: RDREC-COPY
- **In general, the assembler cannot determine whether or not the expression is legal at assembly time. This work will be handled by a linking loader.**

Assembler Design Options

One-pass assemblers

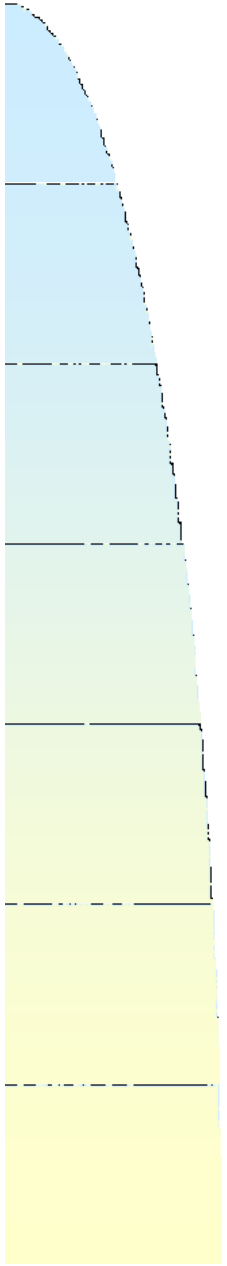
Multi-pass assemblers

Two-pass assembler with overlay structure

Two-Pass Assembler with Overlay Structure

- **For small memory**
  - pass 1 and pass 2 are never required at the same time
  - three segments
    - root: driver program and shared tables and subroutines
    - pass 1
    - pass 2
  - tree structure
  - overlay program

One-Pass Assemblers



- **Main problem**
  - forward references
    - data items
    - labels on instructions
- **Solution**
  - data items: require all such areas be defined before they are referenced
  - labels on instructions: no good solution

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		.			
115		.	SUBROUTINE TO READ RECORD INTO BUFFER		
120		.			
121	2039	INPUT	BYTE	X'F1'	F1
122	203A	MAXLEN	WORD	4096	001000
124		.			
125	203D	RDREC	LDX	ZERO	041006
130	2040		LDA	ZERO	001006
135	2043	RLOOP	TD	INPUT	E02039
140	2046		JEQ	RLOOP	302043
145	2049		RD	INPUT	D82039
150	204C		COMP	ZERO	281006
155	204F		JEQ	EXIT	30205B
160	2052		STCH	BUFFER,X	54900F
165	2055		TIX	MAXLEN	2C203A
170	2058		JLT	RLOOP	382043
175	205B	EXIT	STX	LENGTH	10100C
180	205E		RSUB		4C0000
195		.			
200		.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205		.			
206	2061	OUTPUT	BYTE	X'05'	05
207		.			
210	2062	WRREC	LDX	ZERO	041006
215	2065	WLOOP	TD	OUTPUT	E02061
220	2068		JEQ	WLOOP	302065
225	206B		LDCH	BUFFER,X	50900F
230	206E		WD	OUTPUT	DC2061
235	2071		TIX	LENGTH	2C100C
240	2074		JLT	WLOOP	382065
245	2077		RSUB		4C0000
255			END	FIRST	

Figure 2.18 Sample program for a one-pass assembler.

Memory address	Contents			
1000	454F4600	00030000	00xxxxxx	xxxxxxxx
1010	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
•				
•				
•				
2000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxx14
2010	100948--	--00100C	28100630	----48--
2020	--3C2012			
•				
•				
•				

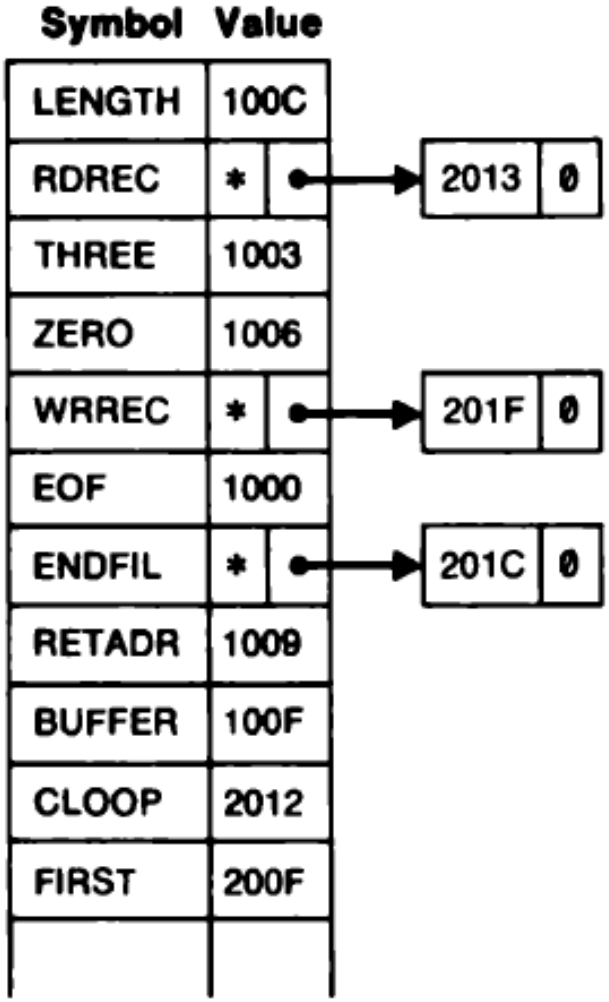


Figure 2.19(a) Object code in memory and symbol table entries for the program in Fig. 2.18 after scanning line 40.

Memory address	Contents			
1000	454F4600	00030000	00xxxxxx	xxxxxxxx
1010	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
•				
•				
•				
2000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxx14
2010	10094820	3D00100C	28100630	202448—
2020	—3C2012	0010000C	100F0010	030C100C
2030	48———08	10094C00	00F10010	00041006
2040	001006E0	20393020	43D82039	28100630
2050	———5490	0F		
•				
•				
•				

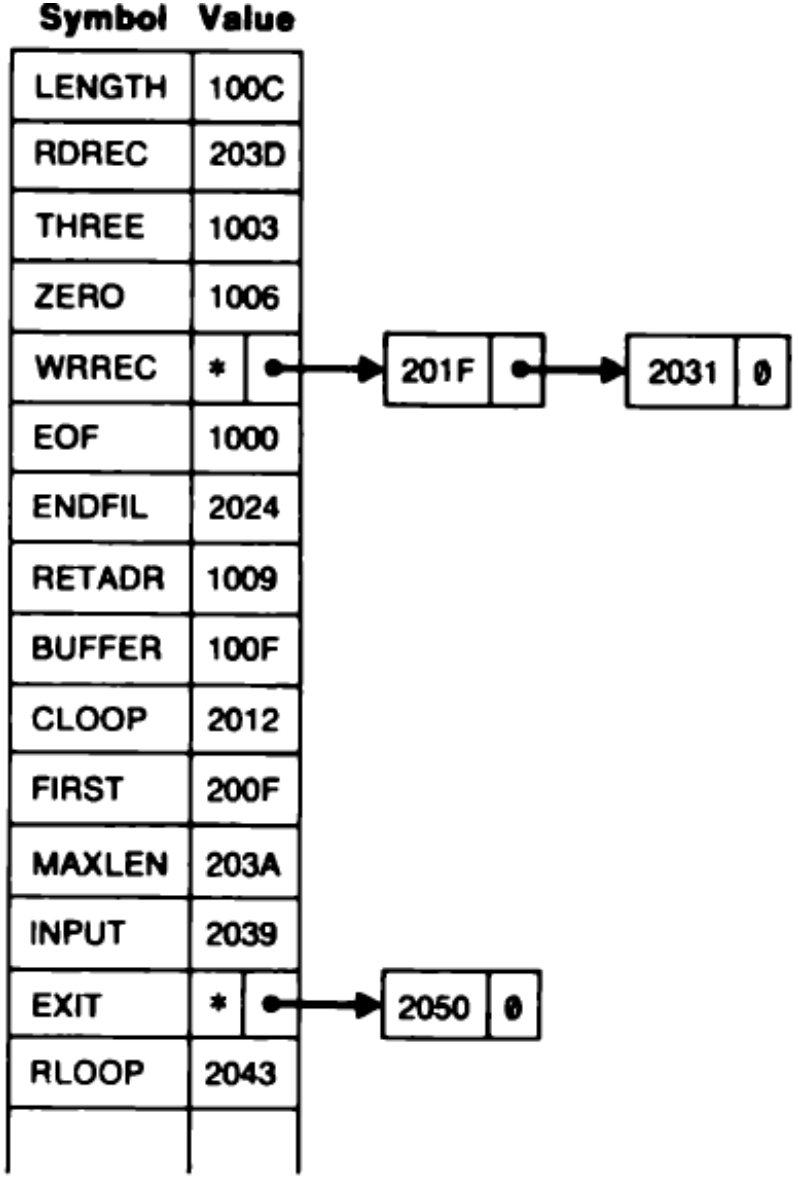


Figure 2.19(b) Object code in memory and symbol table entries for the program in Fig. 2.18 after scanning line 160.

One-Pass Assemblers

- **Main Problem**
  - forward reference
    - data items
    - labels on instructions
- **Two types of one-pass assembler**
  - load-and-go
    - produces object code directly in memory for immediate execution
  - the other
    - produces usual kind of object code for later execution
    - Forward references are written as part of a TEXT RECORD in object program.
    - This correct operand address will be inserted into instruction by loader at load time.

```
HCOPY  001000001007A
T001000094546F46000003000000
T00200F151410094800000100C2810063000004800003C2012
T00201C022024
T00202419001000C100F0010030C100C4800000810094C0000F1001000
T002013022030
T00203D1E041006001006E02039302043D82039281006300000549000F2C203A382043
T002050022058
T00205B071010C4C000005
T00201F022062
T002031022062
T002062180410006E020613020655090FDC20612C100C3820654C0000
E00200F
```

Address of ENDFIL

Figure 2.20 Object program from one-pass assembler for program in Fig. 2.18.

- When program is loaded 2024 will replace previous 0000.
- Other forward references will be handled similar way.
- Literals not allowed.
- Only actual (not relative) addresses in operands.

Load-and-go Assembler

- **Characteristics**
  - Useful for program development and testing
  - Avoids the overhead of writing the object program out and reading it back
  - Both one-pass and two-pass assemblers can be designed as load-and-go.
  - However one-pass also avoids the over head of an additional pass over the source program
  - For a load-and-go assembler, the actual address must be known at assembly time, we can use an absolute program

Forward Reference in One-pass Assembler

- **For any symbol that has not yet been defined**
  1. omit the address translation
  2. insert the symbol into SYMTAB, and mark this symbol undefined
  3. the address that refers to the undefined symbol is added to a list of forward references associated with the symbol table entry
  4. when the definition for a symbol is encountered, the proper address for the symbol is then inserted into any instructions previous generated according to the forward reference list

Load-and-go Assembler (Cont.)

- **At the end of the program**
  - any SYMTAB entries that are still marked with * indicate undefined symbols
  - search SYMTAB for the symbol named in the END statement and jump to this location to begin execution
- **The actual starting address must be specified at assembly time**
- **Example**
  - Figure 2.18, 2.19

Producing Object Code

- **When external working-storage devices are not available or too slow (for the intermediate file between the two passes**
- **Solution:**
  - When definition of a symbol is encountered, the assembler must generate another Tex record with the correct operand address
  - The loader is used to complete forward references that could not be handled by the assembler
  - The object program records must be kept in their original order when they are presented to the loader
- **Example: Figure 2.20**

Multi-Pass Assemblers

- **Restriction on EQU and ORG**
  - no forward reference, since symbols' value can't be defined during the first pass
- **Example**
  - Use link list to keep track of whose value depend on an undefined symbol
- **Figure 2.21**

The reason for this is the symbol definition process in a two-pass assembler. Consider, for example, the sequence

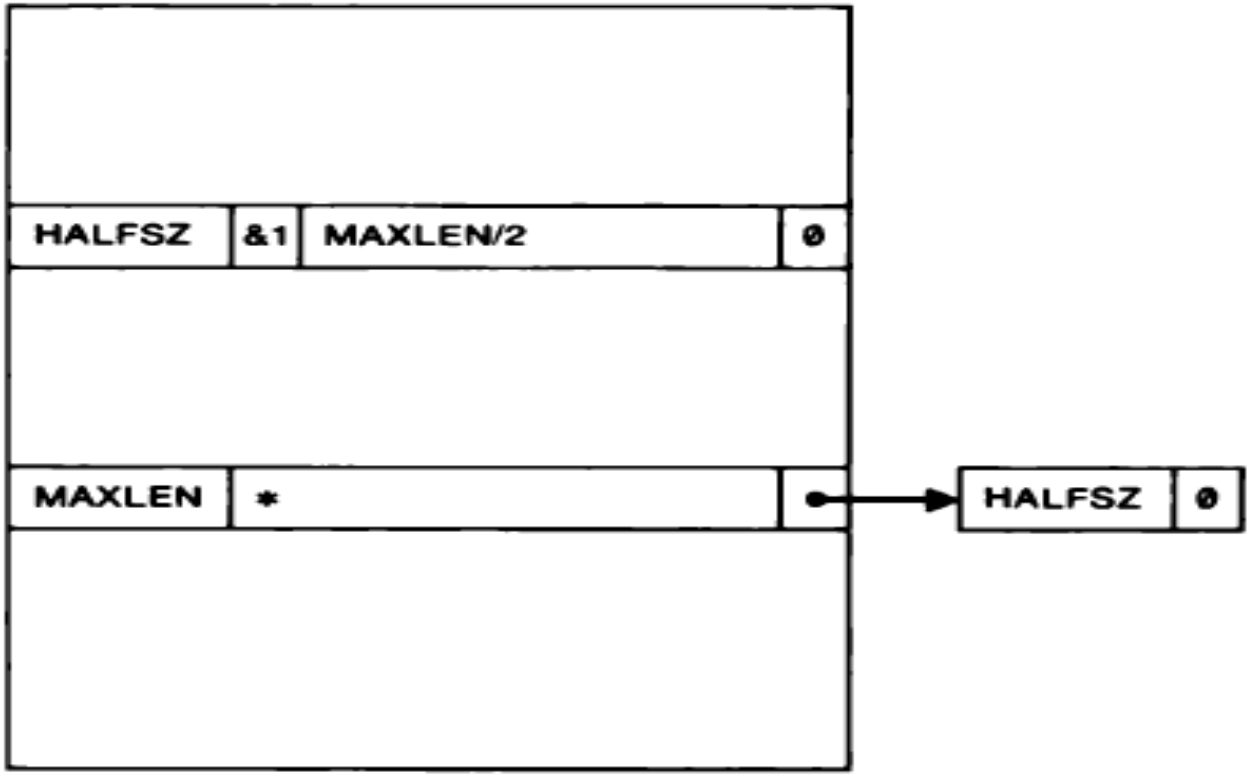
```
ALPHA   EQU     BETA
BETA    EQU     DELTA
DELTA   RESW    1
```

The symbol BETA cannot be assigned a value when it is encountered during the first pass because DELTA has not yet been defined. As a result, ALPHA cannot be evaluated during the second pass. This means that any assembler that makes only two sequential passes over the source program cannot resolve such a sequence of definitions.

- Solution – Forward reference portions are saved in pass 1.
- Additional passes add these informations

```
1       HALFSZ      EQU     MAXLEN/2
2       MAXLEN      EQU     BUFEND-BUFFER
3       PREVBT      EQU     BUFFER-1
                            .
                            .
                            .
4       BUFFER      RESB    4096
5       BUFEND      EQU     *
```

(a)



(b)

Figure 2.21 Example of multi-pass assembler operation.