Assemblers

Assembly language Programming and Assemblers

Module 2 -- Outline

- SIC Programming
- SIC/XE Programming
- Basic Functions of Assembler
- Assembler output format – Header, Text and End Records
- Assembler data structures
- Pass 1 of two pass SIC assembler algorithm
- Pass 2 of two pass SIC assembler algorithm
- Hand assembly of SIC Program

Introduction to Assemblers

Role/Functions of Assembler

- translating mnemonic operation codes to their machine language equivalents
- assigning machine addresses to symbolic labels used by the programmer
- assembler must process statements called assembler directives (or pseudo-instructions)
- assembler must write the generated object code/obj pgm onto some output device which will later be loaded into memory for execution

Role of Assembler

- features and design of an assembler depend heavily upon the source language it translates and the machine language it produces.
- One aspect of this dependence is the existence of different machine instruction formats and codes to accomplish an operation.
- there are some features of an assembler language and assembler that have no direct relation to machine architecture—they are,arbitrary decisions made by the designers of the language.

Example Program Fig. 2.1

Line	Source statement			
5	COPY	START	1000	COPY FILE FROM INPUT TO OUTPUT
10	FIRST	STL	RETADR	SAVE RETURN ADDRESS
15	CLOOP	JSUB	RDREC	READ INPUT RECORD
20		LDA	LENGTH	TEST FOR EOF (LENGTH = 0)
25		COMP	ZERO	
30		JEQ	ENDFIL	EXIT IF EOF FOUND
35		JSUB	WRREC	WRITE OUTPUT RECORD
40		J	CLOOP	LOOP
45	ENDFIL	LDA	EOF	INSERT END OF FILE MARKER
50		STA	BUFFER	
55		LDA	THREE	SET LENGTH = 3
60		STA	LENGTH	
65		JSUB	WRREC	WRITE EOF
70		LDL	RETADR	GET RETURN ADDRESS
75		RSUB		RETURN TO CALLER
80	EOF	BYTE	C'EOF'	
85	THREE	WORD	3	
90	ZERO	WORD	0	
95	RETADR	RESW	1	
100	LENGTH	RESW	1	LENGTH OF RECORD
105	BUFFER	RESB	4096	4096-BYTE BUFFER AREA
110				

Example Program Fig. 2.1

```
.
.          SUBROUTINE TO READ RECORD INTO BUFFER
.
RDREC      LDX      ZERO          CLEAR LOOP COUNTER
           LDA      ZERO          CLEAR A TO ZERO
RLOOP      TD       INPUT         TEST INPUT DEVICE
           JEQ      RLOOP         LOOP UNTIL READY
           RD       INPUT         READ CHARACTER INTO REGISTER A
           COMP     ZERO          TEST FOR END OF RECORD (X'00')
           JEQ      EXIT          EXIT LOOP IF EOR
           STCH     BUFFER,X      STORE CHARACTER IN BUFFER
           TIX      MAXLEN        LOOP UNLESS MAX LENGTH
           JLT      RLOOP           HAS BEEN REACHED
EXIT       STX      LENGTH        SAVE RECORD LENGTH
           RSUB                   RETURN TO CALLER
INPUT      BYTE     X'F1'         CODE FOR INPUT DEVICE
MAXLEN     WORD     4096
```

Example Program Fig. 2.1

```
200     .          SUBROUTINE TO WRITE RECORD FROM BUFFER
205     .
210     WRREC      LDX      ZERO          CLEAR LOOP COUNTER
215     WLOOP      TD       OUTPUT        TEST OUTPUT DEVICE
220                JEQ      WLOOP         LOOP UNTIL READY
225                LDCH     BUFFER,X      GET CHARACTER FROM BUFFER
230                WD       OUTPUT        WRITE CHARACTER
235                TIX      LENGTH        LOOP UNTIL ALL CHARACTERS
240                JLT      WLOOP           HAVE BEEN WRITTEN
245                RSUB                   RETURN TO CALLER
250     OUTPUT     BYTE     X'05'         CODE FOR OUTPUT DEVICE
255                END      FIRST
```

Figure 2.1 Example of a SIC assembler language program.

Parts

Label

Mnemonic

Operand

Comment

Example Program (Fig. 2.1)

- **Purpose**
  - Prgram contains a main routine that reads records from input device identified with a device code F1 and copies them to output device (code 05)
  - main routine calls subroutine RDREC to read a record into a buffer and subroutine WRREC to write the record from buffer to output device
  - Each subroutine must transfer the record one character at a time because the only I/O instructions available are RD and WD.
  - The buffer is necessary because the I/O rates for the two devices(eg: disk and a slow printing terminal)may be very different

Example Program (Fig. 2.1)

- **Purpose**
  - The end of each record is marked with a null character(hexadecimal 00)
  - If a record is longer than the length of the buffer (4096 bytes),only the first 4096 bytes are copied
  - The end of the file to be copied is indicated by a zero-length record.
  - When the end of file is detected,the program writes EOF on the output device and terminates by executing an RSUB instruction
  - We assume that this program was called by the operating system using a JSUB instruction,RSUB will return control to the operating system

Figure 2.1 (Pseudo code)

```
Program copy {
            save return address;
 cloop:   call subroutine RDREC to read one record;
           if length(record)=0 {
               call subroutine WRREC to write EOF;
          }  else {
               call subroutine WRREC to write one record;
               goto cloop;
           }
           load return address
           return to caller
 }
```

An Example (Figure 2.1, Cont.)

EOR:
character x‘00’

```
Subroutine RDREC {
            clear A, X register to 0;
 rloop:     read character from input device to A register
            if not EOR {
                    store character into buffer[X];
                    X++;
                    if X < maximum length
                         goto rloop;
            }
            store X to length(record);
            return
 }
```

An Example (Figure 2.1, Cont.)

```
Subroutine WDREC {
            clear X register to 0;
 wloop:   get character from buffer[X]
          write character from X to output device
          X++;
          if X < length(record)
               goto wloop;
          return
 }
```

- **Example Program & Object Code**

- **A Simple SIC Assembler**
- program(fig 2.2) with the generated object code for each statement.
- The column headed Loc gives the machine address(in hexadecimal)for each part of the assembled program.
- We have assumed that the program starts at address 1000.

Basic SIC Assembler

- **The translation of source program to object code requires us to accomplish the following functions (not necessarily in the order given):**
- 1.Convert mnemonic operation codes to their machine language equivalents—e.g.,translate STL to 14(line 10).
- 2.Convert symbolic operands to their equivalent machine addresses—e.g.,translate RETADR to 1033 (line 10).
- 3.Build the machine instructions in the proper format.
- 4.Convert the data constants specified in the source program into their internal machine representations—e.g.,translate EOF to 454F46 (line 80).
- 5.Write the object program and the assembly listing.

Basic SIC Assembler

- All of these functions except number 2 can easily be accomplished by sequential processing of the source program,one line at a time.
- **The translation of addresses presents a problem.**
- **Consider the statement**
- **10 1000 FIRST STL RETADR 141033**
- **This instruction contains a forward reference—that is,a reference to a label(RETADR)that is defined later in the program.**
- **If we attempt to translate the program line by line,we will be unable to process this statement because we do not know the address that will be assigned to RETADR.**

Basic SIC Assembler

- Because of this,most assemblers make two passes over the source program.
- The **first pass** scan the source program for label definitions and assign addresses
- The **second pass** performs most of the actual translation previously described.

Assembler Directives

- **Pseudo-Instructions**
  - Not translated into machine instructions
  - Providing information to the assembler

Assembler Directives

- **START**-specify name & starting address
- **END**-end of source program, specify the first execution instruction
- **BYTE**-generate **character** or hexadecimal constant occupying as many bytes as needed to represent the constant
- **WORD**- generate 1 word **integer** constant
- **RESB**-reserve the indicated number of bytes for a data area
- **RESW**-reserve the indicated number of words for a data area

Basic SIC Assembler

- Examples of assembler directives are statements like BYTE and WORD,which direct the assembler to generate constants as part of the object program,
- RESB and RESW,which instruct the assembler to reserve memory locations without generating data values.
- The other assembler directives in our sample program are START,which specifies the starting memory address for the object program,and
- END,which marks the end of theprogram.

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	001036
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	081033
75	1027		RSUB		4C0000
80	102A	EOF	BYTE	C'EOF'	454F46
85	102D	THREE	WORD	3	000003
90	1030	ZERO	WORD	0	000000
95	1033	RETADR	RESW	1	
100	1036	LENGTH	RESW	1	
105	1039	BUFFER	RESB	4096	

```
110                 .
115                 .          SUBROUTINE TO READ RECORD INTO BUFFER
120                 .
125     2039        RDREC      LDX      ZERO          041030
130     203C                   LDA      ZERO          001030
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          001000
195                 .
```

```
200                 .          SUBROUTINE TO WRITE RECORD FROM BUFFER
205                 .
210     2061        WRREC      LDX       ZERO          041030
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'         05
255                            END       FIRST
```

Figure 2.2 Program from Fig. 2.1 with object code.

1036	x	x	x	x	x	x		←	RESW 1
1033	x	x	x	x	x	x		←	RESW 1
1030	0	0	0	0	0	0	0	←	WORD 0
102D	0	0	0	0	0	3	3	←	WORD 3
102A	4	5	4	F	4	6	'E' 'O' 'F'	←	BYTE C'EOF'
1027	4	C	0	0	0	0	RSUB		
1024	0	8	1	0	3	3	LDL RETADR		
				…					
100C	3	0	1	0	1	5	JEQ ENDFIL		
1009	2	8	1	0	3	0	COMP ZERO		
1006	0	0	1	0	3	6	LDA LENGTH		
1003	4	8	2	0	3	9	JSUB RDREC		
1000	1	4	1	0	3	3	STL RETADR		
				…					
0									

Basic SIC Assembler-Example 1

LOCATION COUNTER	LABEL	OPCODE	Operand
	SUM	START	4000(H)
	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

Assign Address

LOCATION COUNTER	LABEL	OPCODE	Operand
	SUM	START	4000(H)
4000	FIRST	LDX	ZERO
4003		LDA	ZERO
4006	LOOP	ADD	TABLE, X
4009		TIX	COUNT
400C		JLT	LOOP
400F		STA	TOTAL
4012		RSUB	
4015	TABLE	RESW	2000
5785	COUNT	RESW	1
5788	ZERO	WORD	0
578B	TOTAL	RESW	1
578E		END	FIRST

Generate Object Code

LOCATION COUNTER	LABEL	OPCODE	Operand	OBJECT CODE
	SUM	START	4000(H)	-
4000	FIRST	LDX	ZERO	045788
4003		LDA	ZERO	005788
4006	LOOP	ADD	TABLE, X	18C015
4009		TIX	COUNT	2C5785
400C		JLT	LOOP	384006
400F		STA	TOTAL	0C578B
4012		RSUB		4C0000
4015	TABLE	RESW	2000	-
5785	COUNT	RESW	1	-
5788	ZERO	WORD	0	000000
578B	TOTAL	RESW	1	-
578E		END	FIRST	-

Copy link

Object Program

- Header Record →only one
- Text Record →any number depends on program length
- End Record → only one

- H^SUM_ _ _^004000^00178E
- T^004000^15^045788^005788^18C015^2C5785^384006^0C578B^4C0000
- T^005788^3^000000
- E^004000

Example -2

line	Label	Opcode	Operand
1	SUM	START	3000(H)
2	FIRST	LDX	ZERO
3		LDA	THREE
4	LOOP	ADD	TABLE,X
5		TIX	COUNT
6		JLT	LOOP
7		STA	TOTAL
8		RSUB	
9	THREE	WORD	3
10	TABLE	RESW	100
11	COUNT	RESW	20
12	ZERO	WORD	0
13	TOTAL	RESW	1
14		END	FIRST

Assign Address

Loc	Label	Opcode	Operand
	SUM	START	3000(H)
3000	FIRST	LDX	ZERO
3003		LDA	THREE
3006	LOOP	ADD	TABLE,X
3009		TIX	COUNT
300C		JLT	LOOP
300F		STA	TOTAL
3012		RSUB	
3015	THREE	WORD	3
3018	TABLE	RESW	100
3144	COUNT	RESW	20
3180	ZERO	WORD	0
3183	TOTAL	RESW	1
3186		END	FIRST

Generate Object Code

Loc	Label	Opcode	Operand	Object code
	SUM	START	3000(H)	-
3000	FIRST	LDX	ZERO	043180
3003		LDA	THREE	003015
3006	LOOP	ADD	TABLE,X	18B018
3009		TIX	COUNT	2C3144
300C		JLT	LOOP	383006
300F		STA	TOTAL	0C3183
3012		RSUB		4C0000
3015	THREE	WORD	3	000003
3018	TABLE	RESW	100	-
3144	COUNT	RESW	20	-
3180	ZERO	WORD	0	000000
3183	TOTAL	RESW	1	-
3186		END	FIRST	-

Object Program

- H^SUM- - -^003000^000186
- T^003000^18^043180^003015^18B018^2C3144^38 3006^0C3183^4C0000^000003
- T^003180^03^000000
- E^003000

Basic SIC Assembler

- Finally,the assembler must write the generated object code onto some output device.
- This object program will later be loaded into memory for execution.
- The simple object program format we use contains three types of records:**Header,Text,and End**.

Basic SIC Assembler

- The **Header record** contains the program name,starting address,and length.
- **Text records** contain the translated (i.e.,machine code)instructions and data of the program,together with an indication of the addresses where these are to be loaded.
- The **End record** marks the end of the object program and specifies the address in the program where execution is to begin.
- (This is taken from the operand of the program's END statement.
- If no operand is specified,the address of the first executable instruction is used.)

- The formats we use for these records are as follows.
- The details of the formats (column numbers,etc.)are arbitrary
- however,the information contained in these records must be present (in some form) in the object program.

Object Program

- **Header Record**

  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 Record**

  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 Record**

  Col.1 E
  Col.2~7 Address of first executable instruction (hex)
  (END program_name)

Object Program(Fig. 2.3)

```
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
```

- No object code corresponding to addresses 1033-2038
- This storage is reserved by loader for use by program during execution

Two Pass Assembler

- **Pass 1(DEFINE SYMBOLS)**
- Assign addresses to all statements in the program
  - Save the values /addresses assigned to all labels for use in Pass 2
  - Perform some processing of assembler directives
- **Pass 2(ASSEMBLE INSTRUCTIONS &GENERATE OBJECT PROGRAM)**
  - 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

2 Pass Assembler

- Requires 2 scans of the source program.
- Handles forward references easily.
- LC processing is performed in the 1st pass and symbols are stored in the symbol table.
- Second pass Synthesis Target Program.

Design steps of two pass assembler

- Pass I
  1. Separate the assembly statement in to symbol, mnemonic opcode and operand
  2. Build Symbol Table
  3. Perform LC Processing
  4. Construct Intermediate Representation (IR)
- Pass II
  1. Process IR to synthesize the target program

Data Structures

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

Assembler Algorithm and Data Structures

- Our simple assembler uses two major internal data structures:
- the Operation Code Table(OPTAB)
- Symbol Table (SYMTAB).
- OPTAB is used to lookup mnemonic operation codes and translate them to their machine language equivalents.
- In more complex assemblers,this table also contains information about instruction format and length.
- SYMTAB is used to store values (addresses)assigned to labels.

Assembler Algorithm and Data Structures

- Location Counter (LOCCTR) is a variable that is used to help in the assignment of addresses.
- LOCCTR is initialized to the beginning address specified in the START statement.
- After each source statement is processed,the length of the assembled instruction or data area to be generated is added to LOCCTR
- **LOCCTR<---LOCCTR+Instruction Length**
- eg:whenever we reach a label in the source program,the current value of LOCCTR gives the address to be associated with that label.

Two Pass Assembler

- During Pass 1,OPTAB is used to look up and validate operation codes in the source program.
- information from OPTAB in Pass 2 tell us which instruction format to use in assembling the instruction,and any peculiarities of the object code instruction.
- In Pass 2,it is used to translate the operation codes to machine language.
- In our simple SIC assembler,both of these processes could be done together in either Pass 1 or Pass 2.
- However,for a machine (such as SIC/XE)that has instructions of different lengths,we must search OPTAB in the first pass to find the instruction length for incrementing LOCCTR.

Two Pass Assembler

- OPTAB is usually organized as a hash table,with mnemonic operation code as the key.
- (The information in OPTAB is predefined when the assembler itself is written,rather than being loaded into the table at execution time.Instruction set cannot change,it is predefined for an machine architecture)
- OPTAB is a static table—entries are not normally added to or deleted from it.
- The hash table organization provides fast retrieval with a minimum of searching.

Two Pass Assembler-Data Structure

- The symbol table (SYMTAB)includes the name and value(address) for each label in the source program,together with flags to indicate error conditions (e-g.,a symbol defined in two different places).
- This table may also contain other information about the data area or instruction labeled—for example,its type or length.
- During Pass 1 of the assembler,labels are entered into SYMTAB as they are encountered in the source program,along with their as-signed addresses (from LOCCTR).
- During Pass 2,symbols used as operands are looked up in SYMTAB to obtain the addresses to be inserted in the assembled instructions.

Two Pass Assembler-Data Structure

- SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval.
- Since entries are rarely (if ever)deleted from this table,efficiency of deletion is not an important consideration.
- Because SYMTAB is used heavily throughout the assembly,care should be taken in the selection of a hashing function.
- It is a dynamic table.

Two Pass Assembler-Data Structure

- It is possible for both passes of the assembler to read the original source program as input.
- However,there is certain information (such as location counter values and error flags for statements)that can or should be communicated between the two passes.
- For this reason,Pass 1 usually writes an intermediate file that contains each source statement together with its assigned address,error indicators,etc.
- This file is used as the input to Pass 2.
- This working copy of the source program can also be used to retain the results of certain operations that may be performed during Pass 1(scanning operand field for symbols and addressing flags),so these need not be performed again during Pass 2.
- Similarly,pointers into OPTAB and SYMTAB may be retained for each operation code and symbol used.This avoids the need to repeat many of the table-searching operations.

The intermediate file include each source statement, assigned address and error indicator

Source program
Pass 1
Intermediate file
Pass 2
Object codes
OPTAB
SYMTAB
SYMTAB

Pass 1 Algorithm

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	001036
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	081033
75	1027		RSUB		4C0000
80	102A	EOF	BYTE	C'EOF'	454F46
85	102D	THREE	WORD	3	000003
90	1030	ZERO	WORD	0	000000
95	1033	RETADR	RESW	1	
100	1036	LENGTH	RESW	1	
105	1039	BUFFER	RESB	4096	

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	001036
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	081033
75	1027		RSUB		4C0000
80	102A	EOF	BYTE	C'EOF'	454F46
85	102D	THREE	WORD	3	000003
90	1030	ZERO	WORD	0	000000
95	1033	RETADR	RESW	1	
100	1036	LENGTH	RESW	1	
105	1039	BUFFER	RESB	4096	

```
200                 .          SUBROUTINE TO WRITE RECORD FROM BUFFER
205                 .
210     2061        WRREC      LDX       ZERO          041030
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'         05
255                            END       FIRST
```

Figure 2.2 Program from Fig. 2.1 with object code.

Symtab

symbol	value	flags
FIRST	1000	
CLOOP	1003	
ENDFIL	1015	
EOF	102A	
THREE	102D	
ZERO	1030	
RETADR	1033	
LENGTH	1036	

BUFFER	1039	
RDREC	2039	
RLOOP	203F	
EXIT	2057	
INPUT	205D	
MAXLEN	205E	
WRREC	2061	
WLOOP	2064	
OUTPUT	2079	

LDA=00,

LDX=04,

STA=0C,

ADD=18,

TIX=2C,

JLT=38,

RSUB=4C

Line no	LOCATION COUNTER	LABEL	OPCODE	Operand	Object Code
1		SUM	START	4000(H)	-
2	4000	FIRST	LDX	ZERO	045788
3	4003		LDA	ZERO	005788
4	4006	LOOP	ADD	TABLE, X	18C015
5	4009		TIX	COUNT	2C5785
6	400C		JLT	LOOP	384006
7	400F		STA	TOTAL	0C578B
8	4012		RSUB		4C0000
9	4015	TABLE	RESW	2000	-
10	5785	COUNT	RESW	1	-
11	5788	ZERO	WORD	0	000000
12	578B	TOTAL	RESW	1	-
13	578E		END	FIRST	-

SYMTAB

SYMBOL	Address
FIRST	4000
LOOP	4006
TABLE	4015
COUNT	5785
ZERO	5788
TOTAL	578B

OPTAB

Mnemonic	Format	Opcode
ADD m	3/4	18
ADDF m	3/4	58
ADDR r1,r2	2	90
AND m	3/4	40
CLEAR r1	2	B4
COMP m	3/4	28
COMPF m	3/4	88

Pass 1:

```
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 input 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 SYMTAB
    end {if symbol}
search OPTAB for OPCODE
if found then
    add 3 {instruction 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 {if BYTE}
            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}
```

Pass 1

- 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 then written to the intermediate file.
- 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.
- Next it 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 assembler 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 (each word is of size 3bytes so 3*no of words).
- If it is BYTE it adds the length of the constant to the LOCCTR, if RESB it adds number of bytes reserved.
- If it is END directive then it is the end of the program

Pass 1

- Algorithm finds the length of the program by evaluating current LOCCTR minus the starting address mentioned in the operand field of the END directive.

- Each processed line is written to the intermediate file

PASS 1 INPUTS

Input Program		
PGM1	START	1000
	LDA	ALPHA
	MUL	BETA
	STA	GAMMA
ALPHA	WORD	2
BETA	WORD	4
GAMMA	**RESW**	**1**
	END	1000

Opcode Table(OPTAB)	
Opcode	**Hexacode**
LDA	00
MUL	20
STA	0C

PASS 1 OUTPUTS

Intermediate File			
	PGM1	START	1000
1000		LDA	ALPHA
1003		MUL	BETA
1006		STA	GAMMA
1009	ALPHA	WORD	2
100C	BETA	WORD	4
100F	GAMMA	RESW	1

Symbol Table(SymTAB)	
Label Name	**Label Address**
ALPHA	1009
BETA	100C
GAMMA	100F

LOCCTR = 100F

Pass 2 Algorithm

Pass 2:

```
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 OPCODE
```

```
if found then
   begin
      if there is a symbol in OPERAND field then
         begin
            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}
  write last Text record to object program
  write End record to object program
  write last listing line
end {Pass 2}
```

- first input line is read from the intermediate file.
- If the opcode is START,then this line is directly written to the listing file(output file).
- A header record is written in the object program which gives the starting address and the length of the program (which is calculated during pass 1).
- first text record is initialized.
- Comment lines are ignored.
- OPTAB is searched to find the object code of an opcode.
- If there is a symbol in the operand field, the symbol table is searched to get the address value for this which gets appended to the object code of the opcode.
- If the address is not found then zero value is stored as operand's address.
- An error flag is set indicating it as undefined.
- If symbol itself is not found then store 0 as operand address and the object code instruction is assembled.
- If the opcode is BYTE or WORD, then the constant value is converted to its equivalent object code(for example, for character EOF, its equivalent hexadecimal value ’454f46' is stored).
- If the object code cannot fit into the current text record, a new text record is created and the rest of the instructions object code is listed.
- The text records are written to the object program.
- Once the whole program is assemble and when the END directive is encountered, the End record is written

a. Generate the complete object program for the following assembly level program. Also indicate the contents of symbol table at the end. Assume standard SIC model and assume the following machine OPCODES in HEX

LDA →	00	TIX	-	2C
LDX →	04	JLT	-	38
STA →	0C	RSUB	-	4C
ADD →	18			

Source Program

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

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	

Module -3

Assembler features &design options

Contents

- **Machine dependent assembler features-Instruction format and addressing modes, program relocation**
- **Machine Independent assembler features**
- **Literals**
- **Symbol defining statements**
- **Expression**
- **program blocks**
- **Control sections and program Linking.**
- **Assembler design options-**
- **One Pass assembler**
- **Multi pass assembler**
- Implementation example**: MASM Assembler**

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

Machine-dependent Assembler Features

Sec. 2-2

- Instruction formats and addressing modes
- Program relocation

Instruction Format and Addressing Mode

- indirect addressing is indicated by adding the prefix @ to the operand(line no 70)
- Indirect addressing: op @m
- J @RETADR
- Immediate operands are denoted with the prefix #(lines 25,55,133).
- Immediate addressing: op #c
- COMP #0
- LDA #3

Instruction Format and Addressing Mode

- Instructions that refer to memory are normally assembled using either the program-counter relative or the base relative mode.
- The assembler directive BASE(line 13)is used in conjunction with base relative addressing.
- BASE LENGTH

Instruction Format and Addressing Mode

- If the displacements required for both program-counter relative and base relative addressing are too large to fit into a 3-byte instruction,then the 4-byte extended format (Format 4)must be used.
- PC-relative or Base-relative addressing: op m
- The extended instruction format is specified with the prefix + added to the operation code in the source statement (see lines 15,35,65).
- Extended format: +op m
- +JSUB WRREC

Instruction Format and Addressing Mode

- **Differences 1**
- use of register-to-register instructions (in place of register-to-memory instructions)wherever possible.
- For example,the statement on line 150 is changed from COMP ZERO to COMPR A,S.
- Register-to-register instructions are faster than the corresponding register-to-memory operations because they are shorter,and, do not require another memory reference.
- (Fetching an operand from a register is much faster than retrieving it from main memory.)

Instruction Format and Addressing Mode

- **Differences 2**
- immediate and indirect addressing have been used as much as possible (for example,lines 25,55,and 70).
- COMP #0
- J @RETADR
- Likewise,when using immediate addressing,the operand is already present as part of the instruction and need not be fetched from anywhere.
- The use of indirect addressing often avoids the need for another instruction (as in the"return"operation on line 70).

Instruction Format and Addressing Mode

- some of the changes require the addition of other instructions to the program.
- Eg: changing COMP to COMPR on line 150 forces us to add the CLEAR instruction on line 132.
- This still results in an improvement in execution speed.
- The CLEAR is executed only once for each record read,whereas the benefits of COMPR(as opposed to COMP)are realized for every byte of data transferred.

Instruction Format and Addressing Mode

- The larger main memory of SIC/XE means that we may have room to load and run several programs at the same time.
- This kind of sharing of the machine between programs is called multiprogramming which often results in more productive use of the hardware.
- we can load program into memory wherever there is room rather than specifying a fixed address at assembly time

Example SIC/XE program (Figure 2.6, pp.58)

Loc	Source statement			Object code
0000	COPY	START	0	
0000	FIRST	STL	RETADR	17202D
0003		LDB	#LENGTH	69202D
		BASE	LENGTH	
0006	CLOOP	+JSUB	RDREC	4B101036
000A		LDA	LENGTH	032026
000D		COMP	#0	290000
0010		JEQ	ENDFIL	332007
0013		+JSUB	WRREC	4B10105D
0017		J	CLOOP	3F2FEC
001A	ENDFIL	LDA	EOF	032010
001D		STA	BUFFER	0F2016
0020		LDA	#3	010003
0023		STA	LENGTH	0F200D
0026		+JSUB	WRREC	4B10105D
002A		J	@RETADR	3E2003
002D	EOF	BYTE	C'EOF'	454F46
0030	RETADR	RESW	1	
0033	LENGTH	RESW	1	
0036	BUFFER	RESB	4096	

Example SIC/XE program (Figure 2.6, pp.58)

```
        .
        .       SUBROUTINE TO READ RECORD INTO BUFFER
1036    RDREC   CLEAR   X         B410
1038            CLEAR   A         B400
103A            CLEAR   S         B440
103C            +LDY    #4096     75101000
1040    RLOOP   TD      INPUT     E32019
1043            JEQ     RLOOP     332FFA
1046            TD      INPUT     DB2013
1049            COMPR   A,S       A004
104B            JEQ     EXIT      332008
104E            STCH    BUFFER,X  57C003
1051            TIXR    T         B850
1053            JLT     RLOOP     3B2FEA
1056    EXIT    STX     LENGTH    134000
1059            RSUB              4F0000
105C    INPUT   BYTE    X'F1'     F1
```

Example SIC/XE program (Figure 2.6, pp.58)

```
        .
        .       SUBROUTINE TO READ RECORD INTO BUFFER

105D    WRREC   CLEAR   X         B410
105F            LDT     LENGTH    774000
1062    WLOOP   TD      OUTPUT    E32011
1065            JEQ     WLOOP     332FFA
1068            LDCH    BUFFER,X  53C003
106B            WD      OUTPUT    DF2008
106E            TIXR    T         B850
1070            JLT     WLOOP     3B2FEF
1073            RSUB              4F0000
1076    OUTPUT  BYTE    X'05'     05
                END     FIRST
```

START statement

```
5      COPY      START      0
```

- COPY: program name
- START directive specifies a beginning program address
- 0: a relocatable program
  - Tread as if the program is loaded starting at address 0

Register-register instruction

- Convert the mnemonic name to their number equivalents
  - Register name (A, X, L, B, S, T, F, PC, SW) and their values (0,1, 2, 3, 4, 5, 6, 8, 9)
  - May implement in a separate table or preload the register names and values to SYMTAB

Done during pass 2

```
125   CLEAR      X
150   COMPR      A, S
```

****CLEAR A B400

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
- Addressing mode (Refer to Chapter 1)

Mode	Indication	Operand value
Immediate addressing	n=0, i=1,	**TA**
Indirect addressing	n=1, i=0	**((TA))**
Simple addressing	n=0, i=0	Standard SIC
	n=1, i=1	**(TA)**

Translation

- The assembler must in either case,calculate a displacement to be assembled as part of the object instruction.
- This is computed so that the correct target address results when the displacement is added to the contents of the program counter (PC)or the base register(B).
- resulting displacement must be small enough to fit in the 12-bit field in the instruction.
- This means that the displacement must be between 0 and 4095(for base relative mode)or between -2048 and +2047 (for program-counter relative mode).

Translation

- If neither program-counter relative nor base relative addressing can be used(because the displacements are too large),then the 4-byte extended instruction format (Format 4)must be used.
- 4-byte extended instruction format (Format 4) contains a 20-bit address field,which is large enough to contain the full memory address.
- There is no displacement to be calculated in this case.
- For example,in the instruction
- 15 0006 CLOOP +JSUB **RDREC** 4B10**1036**
- the operand address is 1036.
- This full address is stored in the instruction,with bit e set to 1 to indicate extended instruction format.
- programmer must specify the extended format by using the prefix +

Translation

- If extended format is not specified,our assembler first attempts to translate the instruction using program-counter relative addressing
- If this is not possible (because the required displacement is out of range),the assembler then attempts to use base relative addressing.
- If neither form of relative addressing is applicable and extended format is not specified,then the instruction cannot be properly assembled.
- In this case,the assembler must generate an error message.

Translation

- **displacement calculation for program-counter relative and base relative addressing modes.**
- The computation that the assembler needs to perform is essentially the target address calculation in reverse.
- During execution of instructions on SIC (most computers),the program counter is advanced after each instruction is fetched and before it is executed.

Translation-explanation

- during the execution of the STL instruction,PC will contain the address of the next instruction(that is,0003).
- From the Loc column of the listing,we see that RETADR(line 95)is assigned the address 0030.(The assembler will get this address from SYMTAB.)
- The displacement we need in the instruction is 30-3=2D.
- At execution time,the target address calculation performed will be(PC)+disp,resulting in the correct address (0030).
- Note that bit p is set to 1 to indicate program-counter relative addressing,making the last 2 bytes of the instruction 202D.
- Also note that bits n and i are both set to 1,indicating neither indirect nor immediate addressing;this makes the first byte17 instead of 14.

PC-Relative Addressing Modes

- 10 0000 FIRST STL RETADR 17202D

op(6)	n	I	x	b	p	e	disp(12)

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

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

- 40 0017 J CLOOP 3F2FEC

op(6)	n	I	x	b	p	e	disp(12)

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

  - displacement= CLOOP-PC= 6 - 1A= -14= FEC(2's complement of negative number)

Translation

- **In PC relative assembler knows what the contents of the program counter will be at execution time.**
- The base register ,is under control of the programmer,and the programmer must tell the assembler what the base register will contain during execution of the program so that the assembler can compute displacements.
- This is done with the assembler directive BASE.
- The statement BASE LENGTH(line 13)informs the assembler that the base register will contain the address of LENGTH.
- The preceding instruction(LDB #LENGTH)loads this value into the register during program execution.

```
 5    0000    COPY    START   0
10    0000    FIRST   STL     RETADR      17202D
12    0003            LDB     #LENGTH     69202D
13                    BASE    LENGTH
```

Translation

- The assembler assumes for addressing purposes that register B contains this address until it encounters another BASE statement.
- it may be desirable to use register B for another purpose (temporary storage for a data value).
- In such a case,the programmer must use another assembler directive (eg NOBASE) to inform the assembler that the contents of the base register can no longer be used for addressing.
- BASE and NOBASE are assembler directives,and produce no executable code.
- The programmer must provide instructions that load the proper value into the base register during execution.
- if this is not done properly,the target address calculation will not produce the correct operand address

Translation

- EG 1 : 105F LDT LENGTH 774000
- First do with PC
- PC=1062
- TA=PC+Disp
- 0033=1062+disp
- Disp=102F > allowed range
- We need to use Base Relative Addressing
- TA=Base+Disp
- 0033=0033+Disp
- Disp=0
- Another example is the instruction STX LENGTH(line 175).

  Here the displacement calculated is 0.

Base-Relative Addressing Modes

- 12 LDB #LENGTH
- 13 BASE LENGTH
- 160 104E STCH BUFFER, X 57C003

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

op(6)	n	I	x	b	p	e	disp(12)

(54) 1 1 1 0 1 0 0036-1051= $-101B_{16}$

- ☞ 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

Translation

- According to the BASE statement,register B will contain 0033 (the address of LENGTH)during execution.
- The address of BUFFER is 0036.
- Thus the displacement in the instruction is 36-33=3.
- Notice that bits x and b are set to 1 in the assembled instruction to indicate indexed and base relative addressing.

Translation

- difference between the assembly of the instructions on lines 20 and 175.
- On line 20,LDA LENGTH is assembled with program-counter relative addressing.
- On line 175,STX LENGTH uses base relative addressing
- (If you calculate the program-counter relative displacement that would be required for the statement on line 175,you will see that it is too large to fit into the 12-bit displacement field.)
- The statement on line 20 could also have used base relative mode.
- In our assembler,program-counter relative ,is arbitrarily chosen to attempt first.

Translation

- The assembly of an instruction that specifies **immediate addressing** is simpler because no memory reference is involved.
- convert the immediate operand to its internal representation and insert it into the instruction.
- 55 0020 LDA #3 010003
- operand stored in the instruction as 003,and bit i set to 1 to indicate immediate addressing.
- 133 103C +LDT #4096 75101000
- operand(4096) is too large to fit into the 12-bit displacement field,so the extended instruction format is called for.
- (If the operand were too large even for this 20-bit address field,immediate addressing could not be used.)

Immediate Address Translation

- 55 0020 LDA #3 010003

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

op(6)	n	I	x	b	p	e	disp(12)

- 133 103C +LDT #4096 75101000

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

op(6)	n	I	x	b	p	e	disp(20)

Translation

- A different way of using immediate addressing is
- 12 0003 LDB #LENGTH 69202D
- the immediate operand is the symbol LENGTH.
- Since the value of this symbol is the address assigned to it,this immediate instruction has the effect of loading register B with the address of LENGTH.
- combined program-counter relative addressing with immediate addressing.
- if immediate mode is specified,the target address (not the contents stored at that address)becomes the operand.

Immediate Address Translation (Cont.)

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

op(6)	n	I	x	b	p	e	disp(12)

$(68)_{16}$ 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

Translation

- The assembly of instructions that specify **indirect addressing** displacement is computed to produce the target address
- Then bit n is set to indicate that the contents stored at this location represent the address of the operand,not the operand itself.
- Line 70 shows a statement that combines program-counter relative and indirect addressing in this way.

Indirect Address Translation

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

op(6)	n	I	x	b	p	e	disp(12)

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

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

Program Relocation

- More then one program may share memory and resources at a time
- if we knew in advance exaclty which program is to be executed concurrently we could assign addresses when program is assembled so that programs fit together with no overlapping/space waste
- Not practical to plan program execution as dont know exaclty when jobs will be submitted and how long they will run
- So it is desirable to load program into memory wherever there is space and starting address of program is also not known until load time
-

Line	Source statement			
5	COPY	START	1000	COPY FILE FROM INPUT TO OUTPUT
10	FIRST	STL	RETADR	SAVE RETURN ADDRESS
15	CLOOP	JSUB	RDREC	READ INPUT RECORD
20		LDA	LENGTH	TEST FOR EOF (LENGTH = 0)
25		COMP	ZERO	
30		JEQ	ENDFIL	EXIT IF EOF FOUND
35		JSUB	WRREC	WRITE OUTPUT RECORD
40		J	CLOOP	LOOP
45	ENDFIL	LDA	EOF	INSERT END OF FILE MARKER
50		STA	BUFFER	
55		LDA	THREE	SET LENGTH = 3
60		STA	LENGTH	
65		JSUB	WRREC	WRITE EOF
70		LDL	RETADR	GET RETURN ADDRESS
75		RSUB		RETURN TO CALLER
80	EOF	BYTE	C'EOF'	
85	THREE	WORD	3	
90	ZERO	WORD	0	
95	RETADR	RESW	1	
100	LENGTH	RESW	1	LENGTH OF RECORD
105	BUFFER	RESB	4096	4096-BYTE BUFFER AREA
110	.			
115	.	SUBROUTINE TO READ RECORD INTO BUFFER		

Example Fig. 2.1

Program Relocation

- **Example** Fig. 2.1
  - *Absolute program*, starting address 1000(specified at assembly time)

    e.g. 55 101B LDA THREE 00102D

This statement is translated as 00102D,

Register A is to be loaded from memory address 102D

Program Relocation

- Suppose try to load and execute the program to address 2000(Relocate) instead of 1000

  e.g. 55 101B LDA THREE
  00202D

- Each Absolute address should be modified so that we can load and execute at address 2000
- 102D will be probably some other users program

Program Relocation

- Since the assembler does not know the actual location where the program will be loaded,it cannot make necessary changes in the address used by the program
- Assembler can identify for the loader those parts of object program that needs modification
- An object program that contains information necessary to perform this kind of modification is called relocatable program

Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	17202D
12	0003		LDB	#LENGTH	69202D
13			BASE	LENGTH	
15	0006	CLOOP	+JSUB	RDREC	4B101036
20	000A		LDA	LENGTH	032026
25	000D		COMP	#0	290000
30	0010		JEQ	ENDFIL	332007
35	0013		+JSUB	WRREC	4B10105D
40	0017		J	CLOOP	3F2FEC
45	001A	ENDFIL	LDA	EOF	032010
50	001D		STA	BUFFER	0F2016
55	0020		LDA	#3	010003
60	0023		STA	LENGTH	0F200D
65	0026		+JSUB	WRREC	4B10105D
70	002A		J	@RETADR	3E2003
80	002D	EOF	BYTE	C'EOF'	454F46
95	0030	RETADR	RESW	1	
100	0033	LENGTH	RESW	1	
105	0036	BUFFER	RESB	4096	
110		.			
115		.	SUBROUTINE TO READ RECORD INTO BUFFER		

Program Relocation

- Assembled the program using starting address 0000
- (a) shows how it is loaded in memory at address 0000

SIC XE Program

Line No.	Location				Object Code
5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	17202D
12	0003		LDB	#LENGTH	69202D
13			BASE	LENGTH	
15	0006	CLOOP	+JSUB	RDREC	4B101036
	000A		LDA	LENGTH	032026
—					
35	0013		+JSUB	WRREC	4B10105D
40	0017		J	CLOOP	3F2FEC
—					
65	0026		+JSUB	WRREC	4B10105D
—					
100	0036	BUFFER	RESB	4096	

0000	—	
—	—	
0006	4B101036	+JSUB RDREC
—	—	
1036	B410 ←	RDREC
—	—	
1076	—	

5000	—	
—	—	
5006	4B106036	+JSUB RDREC
—	—	
6036	B410 ←	RDREC
—	—	
6076	—	

7420	—	
—	—	
7426	4B108456	+JSUB RDREC
—	—	
8456	B410 ←	RDREC
—	—	
8496	—	

0000	—	
—	—	
0006	4B101036	+JSUB RDREC
—	—	
1036	B410 ←	RDREC
—	—	
1076	—	

5000	—	
—	—	
5006	4B106036	+JSUB RDREC
—	—	
6036	B410 ←	RDREC
—	—	
6076	—	

7420	—	
—	—	
7426	4B108456	+JSUB RDREC
—	—	
8456	B410 ←	RDREC
—	—	
8496	—	

Program Relocation

- **JSUB instruction from line No 15 is loaded at address 0006**
- The address field of this instruction contains 01036(RDREC instruction address)

- Suppose we want to load this program at address 5000
- address of instruction labelled RDREC is then 6036
- so JSUB instruction must be modified to contain this new address
- Suppose if we load the program at address 7420
- JSUB instruction will change to 4B108456 to correspond to new address of RDREC

Irrespective of the storing address loaded, RDREC is always 1036 bytes past the starting address of the program. This means that we can solve the relocation problem in the following way.

- 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. Initialized LOCCTR to 0)
- 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 loader must also be a part of object program**
- **Modification record is used by assembler to inform loader about object codes to be modified at program relocation .**

FORMAT

- 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

0006 CLOOP +JSUB RDREC 4B101036

Opcode 48			n	i	x	b	p	e	Displacement format 4 (20 bits)				
0100	1	0	1	1	0	0	0	1	0000	0001	0000	0011	0110
0100	1011				0001				0000	0001	0000	0011	0110
Address 0006					Address 0007					Address 0008		Address 0009	

The relocation happened from 4B101036 to 4B106036

M∧000007∧05

Line No.	Location				Object Code
5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	17202D
12	0003		LDB	#LENGTH	69202D
13			BASE	LENGTH	
15	0006	CLOOP	+JSUB	RDREC	4B101036
	000A		LDA	LENGTH	032026
—					
35	0013		+JSUB	WRREC	4B10105D
40	0017		J	CLOOP	3F2FEC
—					
65	0026		+JSUB	WRREC	4B10105D
—					
100	0036	BUFFER	RESB	4096	
—					
125	1036	RDREC	CLEAR	X	B410

M∧000007∧05

M∧000014∧05

Object Code

```
H^COPY  ^000000^001077
T^000000^1D^17202D^69202D^4B101036^032026^290000^332007^4B10105D^3F2FEC^032010
T^00001D^13^0F2016^010003^0F200D^4B10105D^3E2003^454F46
T^001036^1D^B410^B400^B440^75101000^E32019^332FFA^DB2013^A004^332008^57C003^B850
T^001053^1D^3B2FEA^134000^4F0000^F1^B410^774000^E32011^332FFA^53C003^DF2008^B850
T^001070^07^3B2FEF^4F0000^05
M^000007^05
M^000014^05
M^000027^05
E^000000
```

圖2.8　相對於圖2.6的目的程式

- Moving the program from one memory location to another location is called as program relocation.
- It is required because
  - More than one program at a time may be sharing memory and other resources.
  - Actual starting address of the program is not known before hand.

Role of assembler:

It can inform the loader that which parts of the object code need to be modified so that relocation can happen successfully.

Program relocation at run time, transfers complete block to some memory locations. This requires as base address and block should be relatively addressed through this base address. This requires both base address and relative address.

- 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

Machine-Independent Assembler Features

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

Literals

- Consider the following example

```
                :
        LDA             FIVE
                :
FIVE    WORD            5
                :
```

- It is convenient to write the value of a constant operand as a part of instruction

```
                :
        LDA     =X'05'
                :
```

Literals

- A literal is identified with the prefix =, followed by a specification of the literal value
- Examples: (Figure 2.10, pp.68)

```
45   001A ENDFIL LDA   =C'EOF'          032010
                                   nixbpe  disp
                                 000000 110010  010
93                     LTORG
     002D *            =C'EOF'          454F46
215  1062 WLOOP   TD   =X'05'           E32011
230  106B         WD   =X'05'           DF2008
     1076 *            =X'05'           05
```

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 ENDFIL LDA =C'EOF' 032010
  - e.g. 215 1062 WLOOP TD =X'05' E32011

Literals vs. Immediate Operands

- **Immediate Operands**
  - The operand value is assembled as <u>part of the machine instruction</u>
  - e.g. 55 0020 LDA #3 010003
- **Literals**
  - The assembler generates the specified value as a constant <u>at some other memory location</u>
  - e.g. 45 001A ENDFIL LDA =C'EOF' 032010

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)

```
210   105D   WRREC   CLEAR   X          B410
212   105F           LDT     LENGTH     774000
215   1062   WLOOP   TD      =X'05'     E32011
220   1065           JEQ     WLOOP      332FFA
225   1068           LDCH    BUFFER,X   53C003
230   106B           WD      =X'05'     DF2008
235   106E           TIXR    T          B850
240   1070           JLT     WLOOP      3B2FEF
245   1073           RSUB               4F0000
255                  END     FIRST
      1076   *      =X'05'              05
```

- 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**

```
COPY     START   0                COPY FILE FROM INPUT TO OUTPUT
FIRST    STL     RETADR           SAVE RETURN ADDRESS
         LDB    #LENGTH           ESTABLISH BASE REGISTER
         BASE    LENGTH
CLOOP   +JSUB    RDREC            READ INPUT RECORD
         LDA     LENGTH           TEST FOR EOF (LENGTH = 0)
         COMP   #0
         JEQ     ENDFIL           EXIT IF EOF FOUND
        +JSUB    WRREC            WRITE OUTPUT RECORD
         J       CLOOP            LOOP
ENDFIL   LDA    =C'EOF'           INSERT END OF FILE MARKER
         STA     BUFFER
         LDA    #3                SET LENGTH = 3
         STA     LENGTH
        +JSUB    WRREC            WRITE EOF
         J      @RETADR           RETURN TO CALLER
         LTORG
RETADR   RESW    1
LENGTH   RESW    1                LENGTH OF RECORD
BUFFER   RESB    4096             4096-BYTE BUFFER AREA
BUFEND   EQU     *
MAXLEN   EQU     BUFEND-BUFFER    MAXIMUM RECORD LENGTH
```

Use = to represent a literal

```
5    0000   COPY     START   0
10   0000   FIRST    STL     RETADR    17202D
13   0003            LDB     #LENGTH   69202D
14                   BASE    LENGTH
15   0006   CLOOP   +JSUB    RDREC     4B101036
20   000A            LDA     LENGTH    032026
25   000D            COMP   #0         290000
30   0010            JEQ     ENDFIL    332007
35   0013           +JSUB    WRREC     4B10105D
40   0017            J       CLOOP     3F2FEC
45   001A   ENDFIL   LDA    =C'EOF'    032010
50   001D            STA     BUFFER    0F2016
55   0020            LDA    #3         010003
60   0023            STA     LENGTH    0F200D
65   0026           +JSUB    WRREC     4B10105D
70   002A            J      @RETADR    3E2003
93                   LTORG
     002D   *       =C'EOF'            454F46
95   0030   RETADR   RESW    1
100  0033   LENGTH   RESW    1
105  0036   BUFFER   RESB    4096
```

If we do not use a LTORG on line 93, the literal =C'EOF' will be placed at the end of the program.

This operand will then begin at address 1073 and be too far away from the instruction that uses it. PC-relative addressing mode cannot be used.

LTORG thus is used when we want to keep the literal operands close to the instruction that uses it.

LTORG

- The Literal Pool contains a collection of anonymous constant definitions, which are generated by the assembler.
- The LTORG directive defines the start of a literal pool.
- Here, line 243 shows a literal that is inserted by the assembler.
- Note that lines 242 and 243 are listed at the same address (hexadecimal 000308). The directive generates no code

.

```
                    240 *    LITERAL POOL
                    241 ********************************
000308              242          LTORG *
000308 00000001     243                =F'1'
000000              244          END   LAB1
```

The creation of a literal pool gives the following advantages:

- Automatic organization of the literal data into sections that are correctly aligned and arranged so that minimal space is wasted in the literal pool.

- Assembling of duplicate data into the same area.

- Because all literals are cross-referenced, you can find the literal constant in the pool into which it has been assembled.

```
5     COPY      START   0                 COPY FILE FROM INPUT TO OUTPUT
10    FIRST     STL     RETADR            SAVE RETURN ADDRESS
13              LDB     #LENGTH           ESTABLISH BASE REGISTER
14              BASE    LENGTH
15    CLOOP    +JSUB    RDREC             READ INPUT RECORD
20              LDA     LENGTH            TEST FOR EOF (LENGTH = 0)
25              COMP    #0
30              JEQ     ENDFIL            EXIT IF EOF FOUND
35             +JSUB    WRREC             WRITE OUTPUT RECORD
40              J       CLOOP             LOOP
45    ENDFIL    LDA     =C'EOF'           INSERT END OF FILE MARKER
50              STA     BUFFER
55              LDA     #3                SET LENGTH = 3
60              STA     LENGTH
65             +JSUB    WRREC             WRITE EOF
70              J       @RETADR           RETURN TO CALLER
93              LTORG
95    RETADR    RESW    1
100   LENGTH    RESW    1                 LENGTH OF RECORD
105   BUFFER    RESB    4096              4096-BYTE BUFFER AREA
106   BUFEND    EQU     *
107   MAXLEN    EQU     BUFEND-BUFFER     MAXIMUM RECORD LENGTH
```

Use = to represent a literal

original program

5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	17202D
12	0003		LDB	#LENGTH	69202D
13			BASE	LENGTH	
15	0006	CLOOP	+JSUB	RDREC	4B101036
20	000A		LDA	LENGTH	032026
25	000D		COMP	#0	290000
30	0010		JEQ	ENDFIL	332007
35	0013		+JSUB	WRREC	4B10105D
40	0017		J	CLOOP	3F2FEC
45	001A	ENDFIL	LDA	EOF	032010
50	001D		STA	BUFFER	0F2016
55	0020		LDA	#3	010003
60	0023		STA	LENGTH	0F200D
65	0026		+JSUB	WRREC	4B10105D
70	002A		J	@RETADR	3E2003
80	002D	EOF	BYTE	C'EOF'	454F46
95	0030	RETADR	RESW	1	
100	0033	LENGTH	RESW	1	
105	0036	BUFFER	RESB	4096	
110					

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
    - 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

Comparing the character string defining them

- For example, =X'05' and =X'05'

Comparing the generated data value

- For example, =C'EOF' and =X'454F46'
- More intelligent
- However, usually the benefit is not great enough to justify the added complexity.

```
205          .
210    105D  WRREC   CLEAR   X          B410
212    105F          LDT     LENGTH     774000
215    1062  WLOOP   TD      =X'05'     E32011
220    1065          JEQ     WLOOP      332FFA
225    1068          LDCH    BUFFER,X   53C003
230    106B          WD      =X'05'     DF2008
235    106E          TIXR    T          B850
240    1070          JLT     WLOOP      3B2FEF
245    1073          RSUB               4F0000
255                  END     FIRST
       1076  *      =X'05'              05
```

Most assembler can recognize duplicate literals.

- E.g., There are two uses of =X'05' on lines 215 and 230 respectively.
- However, just one copy is generated on line 1076.

Literal - Implementation (3/3)

- **LITTAB**
  - literal name, the operand value and length, the address assigned to the operand
  - Organized as Hash Table

LITTAB

Literal	Hex Value	Length	Address
C'EOF'	454F46	3	002D
X'05'	05	1	1076

How the Assembler Handles Literals?

- **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

Pass 1:

- When encountering an literal, try to find it in LITTAB. If it can be found, do nothing.
- Otherwise, the literal is added to the LITTAB
  - Name, operand value and length can be entered now
  - Only the address field is left blank.
- When encountering a LTORG or the end of the program, the assembler makes a scan of LITTAB. At this time, each literal currently in this table is assigned an address. The location counter then should be updated to reflect the number of bytes occupied by each literal.

Pass 2:

- For each literal operand encountered, we search it in LITTAB and find its assigned address.
- The data values specified by the literals in each literal pool are inserted at the appropriate places in the object program exactly as if these values had been generated by BYTE or WORD statements.

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
EXIT	1056
INPUT	105C
WREC	105D
WLOOP	1062

LITTAB

Literal	Hex Value	Length	Address
C'EOF'	454F46	3	002D
X'05'	05	1	1076

Symbol-Defining Statements

- Users can define *labels* on instructions or data areas
  - The value of a label is the address assigned to the statement
- Users can also define *symbols* with values

```
symbol          EQU          value
```

  - **`value`** can be constants, other symbols, expressions
  - Making source program easier to understand
  - No forward reference

Symbol-Defining Statements

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

Example 2

Define mnemonic names for registers

- `A  EQU  0`
- `X  EQU  1`

```
RMO  A,X
```

RMO 0, 1

- **Example 3 (general purpose registers)**
  - BASE EQU R1
  - COUNT EQU R2
  - INDEX EQU R3
- **Example 4**
  - MAXLEN EQU BUFEND-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**

- Assembler directive **ORG**
  - Allow the assembler to reset the PC to values

    Syntax: ORG value
  - When ORG is encountered, the assembler resets its LOCCTR to the specified value
  - ORG will affect the values of all labels defined until the next ORG

Forward-Reference Problem

- Forward reference is not allowed for either EQU or ORG.
  - All terms in the value field must have been defined previously in the program.
  - The reason is that all symbols must have been defined during Pass 1 in a two-pass assembler.
    - Allowed:

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

    - Not allowed:

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

Forward-Reference Problem

```
        ORG     ALPHA
BYTE1   RESB    1
BYTE2   RESB    1
BYTE3   RESB    1
        ORG
ALPHA   RESB    1
```

Not allowed

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

Not allowed

- **Assembler would not know during pass1 what value to assign to locctr in response to first ORG statement.**
- **Symbols BYTE 1,BYTE 2 ,BYTE 3 could not be assigned values during Pass 1**

-Expression

- The assemblers allow “the use of expressions as operand”
  - The assembler evaluates the expressions and produces a single operand address or value
  - Expressions consist of
    - Operator
      - +,-,*,/ (division is usually defined to produce an integer result)
    - Individual terms
      - Constants
      - User-defined symbols
      - Special terms, e.g., *, the current value of LOCCTR
  - Examples
    - MAXLEN EQU BUFEND-BUFFER
    - STAB RESB (6+3+2)*MAXENTRIES

each

as

- Values of terms can be
  - Absolute (independent of program location)
    - constants
  - Relative (to the beginning of the program)
    - Address labels
    - * (value of LOCCTR)
- Expressions can be
  - Absolute
    - Only absolute terms
      - `MAXLEN  EQU  1000`
    - Relative terms in pairs with opposite signs for each pair
      - `MAXLEN  EQU  BUFEND-BUFFER`
  - Relative
    - All the relative terms except one can be paired as described in "absolute". The remaining unpaired relative term must have a positive sign.
      - `STAB  EQU  OPTAB + (BUFEND - BUFFER)`

Absolute Expression

- Relative term or expression implicitly represents (S+r)
  - S: the starting address of the program
  - r: value of the term or expression relative to S
- For example
  - BUFFER: S+r1
  - BUFEND: S+r2
- The expression, BUFEND-BUFFER, is absolute.
  - MAXLEN = (S+r2)-(S+r1) = r2-r1 (no S here)
  - MAXLEN means the length of the buffer area
- Illegal expressions: BUFEND+BUFFER, 100-BUFFER, 3*BUFFER

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 <u>relative terms</u> are <u>paired with opposite signs</u>, the dependency on the program starting address is canceled out; the result is an absolute value**

- if starting address is 0000 then,

BUFFER=0050 and BUFFEND=0055, MAXLEN=0055-0050 = 05

- if starting address is 1000 then,

BUFFER=1050 and BUFFEND=1055, MAXLEN=1055-1050 = 05

In both the above conditions the value of MAXLEN is same i.e., 05. hence MAXLEN is absolute expression.

Restriction of Relative Expressions

- No relative terms may enter into a multiplication or division operation
  - 3 * BUFFER
- Expressions that do not meet the conditions of either “absolute” or “relative” should be flagged as errors.
  - BUFEND + BUFFER
  - 100 - BUFFER

- To determine the type of an expression, we must keep track of the types of all symbols defined in the program.
- We need a “flag” in the SYMTAB for indication.

Symbol	Type	Value
RETADR	R	0030
BUFFER	R	0036
BUFEND	R	1036
MAXLEN	A	1000

Program Blocks

- **Program blocks**
  - refer to segments of code that are rearranged within a single object program unit
  - **USE [blockname]**

Program Blocks

- Each program block may actually contain several separate segments of the source program.
- The assembler will logically rearrange these segments to gather together the pieces of each block.
- The result is the same as if the programmer had physically rearranged the source statements to group together all the source lines belonging to each block.

- As an *example*, 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
- The assembler directive USE indicates which portions of the source program belong to the various blocks.

- 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

Program with Multiple Program Blocks

At the beginning, the default block is assumed.

```
COPY     START   0            COPY FILE FROM INPUT TO OUTP
FIRST    STL     RETADR       SAVE RETURN ADDRESS
CLOOP    JSUB    RDREC        READ INPUT RECORD
         LDA     LENGTH       TEST FOR EOF (LENGTH = 0)
         COMP    #0
         JEQ     ENDFIL       EXIT IF EOF FOUND
         JSUB    WRREC        WRITE OUTPUT RECORD
         J       CLOOP        LOOP
ENDFIL   LDA     =C'EOF'      INSERT END OF FILE MARKER
         STA     BUFFER
         LDA     #3           SET LENGTH = 3
         STA     LENGTH
         JSUB    WRREC        WRITE EOF
         J       @RETADR      RETURN TO CALLER
         USE     CDATA
RETADR   RESW    1
LENGTH   RESW    1            LENGTH OF RECORD
         USE     CBLKS
BUFFER   RESB    4096         4096-BYTE BUFFER AREA
BUFEND   EQU     *            FIRST LOCATION AFTER BUFFER
MAXLEN   EQU     BUFEND-BUFFER MAXIMUM RECORD LENGTH
```

Resume the default block

```
.
.          SUBROUTINE TO READ RECORD INTO BUFFER
.
           USE
RDREC      CLEAR     X                 CLEAR LOOP COUNTER
           CLEAR     A                 CLEAR A TO ZERO
           CLEAR     S                 CLEAR S TO ZERO
          +LDT      #MAXLEN
RLOOP      TD        INPUT             TEST INPUT DEVICE
           JEQ       RLOOP             LOOP UNTIL READY
           RD        INPUT             READ CHARACTER INTO REGISTER A
           COMPR     A,S               TEST FOR END OF RECORD (X'00')
           JEQ       EXIT              EXIT LOOP IF EOR
           STCH      BUFFER,X          STORE CHARACTER IN BUFFER
           TIXR      T                 LOOP UNLESS MAX LENGTH
           JLT       RLOOP               HAS BEEN REACHED
EXIT       STX       LENGTH            SAVE RECORD LENGTH
           RSUB                        RETURN TO CALLER
           USE       CDATA
INPUT      BYTE      X'F1'             CODE FOR INPUT DEVICE
```

Resume the CDATA block

```
195     .
200     .          SUBROUTINE TO WRITE RECORD FROM BUFFER
205     .
208                USE
210     WRREC      CLEAR     X              CLEAR LOOP COUNTER
212                LDT       LENGTH
215     WLOOP      TD       =X'05'          TEST OUTPUT DEVICE
220                JEQ       WLOOP          LOOP UNTIL READY
225                LDCH      BUFFER,X       GET CHARACTER FROM BUFFER
230                WD       =X'05'          WRITE CHARACTER
235                TIXR      T              LOOP UNTIL ALL CHARACTERS
240                JLT       WLOOP            HAVE BEEN WRITTEN
245                RSUB                     RETURN TO CALLER
252                USE       CDATA
253                LTORG
255                END       FIRST
```

Resume the default block

Resume the CDATA block

Why Program Blocks

- To satisfy the contradictive goals:
  - Separate the program into blocks in a particular order
    - Large buffer area is moved to the end of the object program
    - Using the extended format instructions or base relative mode may be reduced. (lines 15, 35, and 65)
    - Placement of literal pool is easier: simply put them before the large data area, CDATA block. (line 253)
  - Data areas are scattered
    - Program readability is better if data areas are placed in the source program close to the statements that reference them.

Loc/Block

5	0000	0	COPY	START	0	
10	0000	0	FIRST	STL	RETADR	172063
15	0003	0	CLOOP	JSUB	RDREC	4B2021
20	0006	0		LDA	LENGTH	032060
25	0009	0		COMP	#0	290000
30	000C	0		JEQ	ENDFIL	332006
35	000F	0		JSUB	WRREC	4B203B
40	0012	0		J	CLOOP	3F2FEE
45	0015	0	ENDFIL	LDA	=C'EOF'	032055
50	0018	0		STA	BUFFER	0F2056
55	001B	0		LDA	#3	010003
60	001E	0		STA	LENGTH	0F2048
65	0021	0		JSUB	WRREC	4B2029
70	0024	0		J	@RETADR	3E203F
92	0000	1		USE	CDATA	
95	0000	1	RETADR	RESW	1	
100	0003	1	LENGTH	RESW	1	
103	0000	2		USE	CBLKS	
105	0000	2	BUFFER	RESB	4096	
106	1000	2	BUFEND	EQU	*	
107	1000		MAXLEN	EQU	BUFEND-BUFFER	
110						

0: default
1: CDATA
2: CBLKS

No block number because MAXLEN is an absolute symbol

```
110                 .
115                 .        SUBROUTINE TO READ RECORD INTO BUFFER
120                 .
123   0027  0                USE
125   0027  0       RDREC    CLEAR    X            B410
130   0029  0                CLEAR    A            B400
132   002B  0                CLEAR    S            B440
133   002D  0               +LDT     #MAXLEN       75101000
135   0031  0       RLOOP    TD       INPUT        E32038
140   0034  0                JEQ      RLOOP        332FFA
145   0037  0                RD       INPUT        DB2032
150   003A  0                COMPR    A,S          A004
155   003C  0                JEQ      EXIT         332008
160   003F  0                STCH     BUFFER,X     57A02F
165   0042  0                TIXR     T            B850
170   0044  0                JLT      RLOOP        3B2FEA
175   0047  0       EXIT     STX      LENGTH       13201F
180   004A  0                RSUB                  4F0000
183   0006  1                USE      CDATA
185   0006  1       INPUT    BYTE     X'F1'        F1
```

```
                    .
                    .          SUBROUTINE TO WRITE RECORD FROM BUFFER
                    .
004D  0                        USE
004D  0             WRREC      CLEAR     X                B410
004F  0                        LDT       LENGTH           772017
0052  0             WLOOP      TD       =X'05'            E3201B
0055  0                        JEQ       WLOOP            332FFA
0058  0                        LDCH      BUFFER,X         53A016
005B  0                        WD       =X'05'            DF2012
005E  0                        TIXR      T                B850
0060  0                        JLT       WLOOP            3B2FEF
0063  0                        RSUB                       4F0000
0007  1                        USE       CDATA
                               LTORG
0007  1             *         =C'EOF                      454F46
000A  1             *         =X'05'                      05
                               END       FIRST
```

	QUIZ	START	**0**
	FIRST	LDA	FIVE
		STA	ALPHA
		LDA	TWO
		STA	BETA
		LDA	ALPHA
		ADD	BETA
		STA	RESULT
	FIVE	WORD	5
	TWO	WORD	2
	ALPHA	RESW	1
	BETA	RESW	1
	RESULT	RESW	1
		END	FIRST

CLEAR= B4, LDA= 00, LDB= 68,

JLT=38, RSUB=4C, ADD=18,

TIX=2C, STA= 0C

	QUIZ	START	**0**	
0000	FIRST	LDA	FIVE	000015
0003		STA	ALPHA	0C001B
0006		LDA	TWO	000018
0009		STA	BETA	0C001E
000C		LDA	ALPHA	00001B
000F		ADD	BETA	18001E
0012		STA	RESULT	0C0021
0015	FIVE	WORD	5	
0018	TWO	WORD	2	
001B	ALPHA	RESW	1	
001E	BETA	RESW	1	
0021	RESULT	RESW	1	
		END	FIRST	

HQUIZ 000000000023
T00000120000150C001B0000180C001E0C0021
E000000

Generate the object program for the following SIC/XE program.

CLEAR= B4, LDA= 00, LDB= 68, JLT=38, RSUB=4C, ADD=18, TIX=2C, STA= 0C

LABEL	OPCODE	OPERAND
SUM	START	0
FIRST	CLEAR	X
	LDA	#0
	+LDB	#TOTAL
	BASE	TOTAL
LOOP	ADD	TABLE, X
	TIX	COUNT
	JLT	LOOP
	STA	TOTAL
	RSUB	
COUNT	RESW	1
TABLE	RESW	2000
TOTAL	RESW	1
	END	FIRST

LOCCTR	LABEL	OPCODE	OPERAND	OBJECT CODE
	SUM	START	0	
0000	FIRST	CLEAR	X	
0002		LDA	#0	
0005		+LDB	#TOTAL	
		BASE	TOTAL	
0009	LOOP	ADD	TABLE, X	
000C		TIX	COUNT	
000F		JLT	LOOP	
0012		STA	TOTAL	
0015		RSUB		
0018	COUNT	RESW	1	
001B	TABLE	RESW	2000	
178B	TOTAL	RESW	1	
178E		END	FIRST	

CLEAR= B4, LDA= 00, LDB= 68, JLT=38, RSUB=4C, ADD=18, TIX=2C, STA= 0C

LOCCTR	LABEL	OPCODE	OPERAND	OBJECT CODE
	SUM	START	0	
0000	FIRST	CLEAR	X	B410
0002		LDA	#0	010000
0005		+LDB	#TOTAL	6910178B
		BASE	TOTAL	
0009	LOOP	ADD	TABLE, X	1BA00F
000C		TIX	COUNT	2F2009
000F		JLT	LOOP	3B2FF7
0012		STA	TOTAL	0F4000
0015		RSUB		4F0000
0018	COUNT	RESW	1	
001B	TABLE	RESW	2000	
178B	TOTAL	RESW	1	
178E		END	FIRST	0000

H^SUM___^000000^178E

T^000000^18^B410^010000^6910178

B^1BA00F^2F2009^3B2FF7^0F4000

^4F0000

E^000000

Length = 178E – 0000 = 178E

Loc	Source statement			Object code
1000	SUM	START	1000	
1000	FIRST	LDA	#0	010000
1003		LDX	#1	050001
1006	LOOP	ADDR	X,A	9010
1008		TIX	#11	2D000B
100B		JLT	LOOP	3B2FF8
100E		END	FIRST	

Disassemble (convert object code back into assembly language) the following SIC/XE program.

HSUM 00100000000E

T0010000E01000005000190102D000B3B2FF8

E001000

Loc	Source statement			Object code
	EXAM2	START	0	
0000	FIRST	LDA	ZERO	00004B
0003		LDX	K128	040048
0006	LOOP	JSUB	GETC	48001E
0009		MUL	K16	200054
000C		STA	ZERO2	0C0058
000F		JSUB	GETC	48001E
0012		ADD	ZERO2	180058
0015		STCH	FIRST,X	548000
0018		TIX	ZERO	2C004B
001B		J	LOOP	3C0006
001E	GETC	TD	DEV	E00057
0021		JEQ	GETC	30001E
0024		RD	DEV	D80057
0027		COMP	K4	280042
002A		JEQ	80	300080

Program Readability

- **Object code**
  - It is not necessary to physically rearrange the generated code in the object program
  - see Fig. 2.13, Fig. 2.14

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

- Pass 1
  - Maintain a separate LOCCTR for each program bloc
    - initialized to 0 when the block is first begun
    - saved when switching to another block
    - restored when resuming a previous block
  - Assign to each label an address relative to the start o the block that contains it
  - Store the block name or number in the SYMTAB along with the assigned relative address of the label
  - Indicate the block length as the latest value of LOCCTR for each block at the end of Pass1
  - Assign to each block a starting address in the object program by concatenating the program blocks in a particular order

- Pass 2
  - Calculate the address for each symbol <u>relative to the start of the object program</u> by adding
    - the location of the symbol relative to the start of its block
    - the assigned starting address of this block

Table for Program Blocks

- At the end of Pass 1:

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

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 of Address Calculation

20	0006	0	LDA	LENGTH	032060

- The value of the operand (LENGTH)
  - Address 0003 relative to Block 1 (CDATA)

    →address 0003+0066=0069 relative to program

    →address 0069-0009=0060 relative to PC, in which the address of PC relative to program is 0009+0000=0009

Example of Address Calculation

```
20  0006  0      LDA    LENGTH      032060
```

- The value of the operand (LENGTH)
  - Address 0003 relative to Block 1 (CDATA)
  - Address 0003+0066=0069 relative to program
  - When this instruction is executed
    - PC = 0000 (starting addr. Of default block) + 0009
  - disp = 0069 - 0009 = 0060
  -

op	nixbpe	disp	
000000	110010	060	=> 032060

SYMTAB

label name	block num	addr.	Flag
LENGTH	1	0003	
....	....	....	....

Object Program

- It is not necessary to physically rearrange the generated code in the object program to place the pieces of each program block together.
- The assembler just simply insert the proper load address in each Text record.

```
PY  000000001071
000001E1720634B2021032060290000332006 4B203B3F2FEE0320550F2056010
001E090F20484B20293E203F
00271DB410B400B44075101000E32038332FFADB2032A00433200857A02FB85
0044093B2FEA13201F4F0000
006C01F1
004D19B410772017E3201B332FFA53A016DF2012B8503B2FEF4F0000
006D04454F4605
00000
```

Source program
Object program
Program loaded in memory
Relative address
Line
5
70
95
100
105
125
180
185
210
245
253
Default(1)
CDATA(1)
CBLKS(1)
Default(2)
CDATA(2)
Default(3)
CDATA(3)
Default(1)
Default(2)
CDATA(2)
Default(3)
CDATA(3)
Default(1)
Default(2)
Default(3)
CDATA(1)
CDATA(2)
CDATA(3)
CBLKS(1)
0000
0027
004D
0066
006C
006D
0071
1070
same order
Not present in object program
Rearrangement through loading

Control Sections and Program Linking

- A Control Section is a part of the program that maintains its identity after assembly
- A Control Section is often used for subroutine or other logical subdivision of a program
- A Control Section can be assembled, loaded, and relocated independently
- Programmer can assemble ,load,manipulate each of these control sections independently

- three controls sections.
  - Assembled together or assembled independently.

Control Sections and Program Linking

- instruction in one control section may need to refer to instructions or data located in another section. i.e.CS is logically related part of pgm
- because of this, there should be some means for linking control sections together
- Fig. 2.15, 2.16

Control Sections and Program Linking

- Program Linking is used to link together logically related control sections
- **Problem:**
- Because the control sections are independenly loaded and relocated
- The assembler does not know where any other control section will be located at execution time.
- When an instruction needs to refer to instructions or data located in another control section,the assembler is unable to process this reference.
-

Control Sections and Program Linking

- Program Linking is used to link together logically related control sections
- **Solution**
- The assembler has to generate information for such kind of references, called external references (References between control sections) , that will allow the loader to perform the required linking.

Line	Source statement			
5	COPY	START	0	COPY FILE FROM INPUT TO OUT
6		EXTDEF	BUFFER,BUFEND,LENGTH	
7		EXTREF	RDREC,WRREC	
10	FIRST	STL	RETADR	SAVE RETURN ADDRESS
15	CLOOP	+JSUB	RDREC	READ INPUT RECORD
20		LDA	LENGTH	TEST FOR EOF (LENGTH = 0)
25		COMP	#0	
30		JEQ	ENDFIL	EXIT IF EOF FOUND
35		+JSUB	WRREC	WRITE OUTPUT RECORD
40		J	CLOOP	LOOP
45	ENDFIL	LDA	=C'EOF'	INSERT END OF FILE MARKER
50		STA	BUFFER	
55		LDA	#3	SET LENGTH = 3
60		STA	LENGTH	
65		+JSUB	WRREC	WRITE EOF
70		J	@RETADR	RETURN TO CALLER
95	RETADR	RESW	1	
100	LENGTH	RESW	1	LENGTH OF RECORD
103		LTORG		
105	BUFFER	RESB	4096	4096-BYTE BUFFER AREA
106	BUFEND	EQU	*	
107	MAXLEN	EQU	BUFEND-BUFFER	

```
109   RDREC    CSECT
110   .
115   .        SUBROUTINE TO READ RECORD INTO BUFFER
120   .
122            EXTREF   BUFFER,LENGTH,BUFEND
125            CLEAR    X                 CLEAR LOOP COUNTER
130            CLEAR    A                 CLEAR A TO ZERO
132            CLEAR    S                 CLEAR S TO ZERO
133            LDT      MAXLEN
135   RLOOP    TD       INPUT             TEST INPUT DEVICE
140            JEQ      RLOOP             LOOP UNTIL READY
145            RD       INPUT             READ CHARACTER INTO REGISTER A
150            COMPR    A,S               TEST FOR END OF RECORD (X'00')
155            JEQ      EXIT              EXIT LOOP IF EOR
160           +STCH     BUFFER,X          STORE CHARACTER IN BUFFER
165            TIXR     T                 LOOP UNLESS MAX LENGTH
170            JLT      RLOOP               HAS BEEN REACHED
175   EXIT    +STX      LENGTH            SAVE RECORD LENGTH
180            RSUB                       RETURN TO CALLER
185   INPUT    BYTE     X'F1'             CODE FOR INPUT DEVICE
190   MAXLEN   WORD     BUFEND-BUFFER
```

```
WRREC     CSECT
.
.         SUBROUTINE TO WRITE RECORD FROM BUFFER
.
          EXTREF    LENGTH,BUFFER
          CLEAR     X                 CLEAR LOOP COUNTER
         +LDT       LENGTH
WLOOP     TD       =X'05'             TEST OUTPUT DEVICE
          JEQ       WLOOP             LOOP UNTIL READY
         +LDCH      BUFFER,X          GET CHARACTER FROM BUFF
          WD       =X'05'             WRITE CHARACTER
          TIXR      T                 LOOP UNTIL ALL CHARACTE
          JLT       WLOOP               HAVE BEEN WRITTEN
          RSUB                        RETURN TO CALLER
          END       FIRST
```

- The assembler etablish a separate locctr(begining with 0)for each control section
- control section differ from PB in that they are handled separately by the assember
- Symbols that are defined in 1 CS may not be used directy by another CS,they are identified as external references for loader to handle
- 2 assembler directives to identify such a references
- EXTREF
- EXTDEF

Assembler Directives for Control Section

- **START:**
- start the first control section
- set program name as the control section name
- define the control section name as an external symbol
- **CSECT:**
- start a new control section
- specify the control section name
- define the control section name as an external symbol

- **EXTDEF: External Definition**
- EXTDEF name [, name]
- external symbols define in this CS and that may be used by other sections
- Control section names do not need to be named in an EXTDEF statement, they are automatically considered to be external symbols
- (In the following, these are COPY, RDREC, and WRREC)
- **EXTREF: External Reference**
- EXTREF name [,name]
- Names symbols that are used in this control section and are defined elsewhere

External Definition and References

- **External definition**
  - EXTDEF names symbols that are defined in this control section and may be used by other sections
- **External reference**
  - EXTREF names symbols that are used in this control section and are defined elsewhere

- External references
  - Instructions in one control section might need to refer to instructions or data located in another section.
- Figure 2.15, multiple control sections.
  - Three sections, main COPY, RDREC, WRREC.
  - Assembler directive CSECT.
  - EXTDEF and EXTREF for external symbols.
  - The order of symbols is not significant.

```
COPY    START    0
        EXTDEF   BUFFER, BUFEND, LENGTH
        EXTREF   RDREC, WRREC
```

External Definition and References

- **EG** .Symbols BUFFER,BUFFEND,LENGTH is defined in CS COPY and made available to other CS
- **Example**
  - 15 0003 CLOOP +JSUB RDREC 4B100000
  - 160 0017 +STCH BUFFER,X 57900000
  - 190 0028 MAXLEN WORD BUFEND-BUFFER 000000

How External Refernces are handled by Assembler

- **Example**
  - **15 0003 CLOOP +JSUB RDREC 4B100000**
- operand RDREC is named in the EXTREF statement for the CS so it is external reference
- Assembler has no idea where CS containg RDREC will be loaded so cannot assemble address for this instruction
- Assembler will insert an address of zero and pass information to loader and loader will prepare proper address and insert at load time
- An extended format instruction must be used to provide space for actual address to be inserted

Figure 2.16, the generated object code.

```
15      0003  CLOOP  +JSUB  RDREC      4B100000
160     0017         +STCH  BUFFER,X   57900000
```

- RDREC is an external reference.
- The assembler has no idea where the control section containing RDREC will be loaded, so it cannot assemble the address.
- The proper address to be inserted at load time.
- Must use extended format instruction for external reference (M records are needed).

```
190     0028  MAXLEN WORD BUFEND-BUFFER
```

- An expression involving two external references.

The Solution

- The assembler inserts an address of zero
- The assembler can only use extended format (Format 4) to provide enough room for address(The relative addressing for external reference is invalid)
- The assembler must remember (via entries in SYMTAB) in which control section a symbol is defined
- The assembler passes information to the loader

Line	Loc	Source statement			Object co
5	0000	COPY	START	0	
6			EXTDEF	BUFFER,BUFEND,LENGTH	
7			EXTREF	RDREC,WRREC	
10	0000	FIRST	STL	RETADR	172027
15	0003	CLOOP	+JSUB	RDREC	4B100000
20	0007		LDA	LENGTH	032023
25	000A		COMP	#0	290000
30	000D		JEQ	ENDFIL	332007
35	0010		+JSUB	WRREC	4B100000
40	0014		J	CLOOP	3F2FEC
45	0017	ENDFIL	LDA	=C'EOF'	032016
50	001A		STA	BUFFER	0F2016
55	001D		LDA	#3	010003
60	0020		STA	LENGTH	0F200A
65	0023		+JSUB	WRREC	4B100000
70	0027		J	@RETADR	3E2000
95	002A	RETADR	RESW	1	
100	002D	LENGTH	RESW	1	
103			LTORG		
	0030	*	=C'EOF'		454F46
105	0033	BUFFER	RESB	4096	
106	1033	BUFEND	EQU	*	
107	1000	MAXLEN	EQU	BUFEND-BUFFER	

```
109     0000    RDREC     CSECT
110             .
115             .         SUBROUTINE TO READ RECORD INTO BUFFE
120             .
122                       EXTREF   BUFFER,LENGTH,BUFEND
125     0000              CLEAR    X                      B410
130     0002              CLEAR    A                      B400
132     0004              CLEAR    S                      B440
133     0006              LDT      MAXLEN                 77201F
135     0009    RLOOP     TD       INPUT                  E3201B
140     000C              JEQ      RLOOP                  332FFA
145     000F              RD       INPUT                  DB2015
150     0012              COMPR    A,S                    A004
155     0014              JEQ      EXIT                   332009
160     0017             +STCH     BUFFER,X               57900000
165     001B              TIXR     T                      B850
170     001D              JLT      RLOOP                  3B2FE9
175     0020    EXIT     +STX      LENGTH                 13100000
180     0024              RSUB                            4F0000
185     0027    INPUT     BYTE     X'F1'                  F1
190     0028    MAXLEN    WORD     BUFEND-BUFFER          000000
```

```
193     0000    WRREC    CSECT
195             .
200             .        SUBROUTINE TO WRITE RECORD FROM BUFF
205             .
207                      EXTREF   LENGTH,BUFFER
210     0000             CLEAR    X                    B410
212     0002            +LDT      LENGTH               77100000
215     0006    WLOOP    TD      =X'05'                E32012
220     0009             JEQ      WLOOP                332FFA
225     000C            +LDCH     BUFFER,X             53900000
230     0010             WD      =X'05'                DF2008
235     0013             TIXR     T                    B850
240     0015             JLT      WLOOP                3B2FEE
245     0018             RSUB                          4F0000
255                      END      FIRST
        001B    *       =X'05'                        05
```

```
160   0017                    +STCH     BUFFER,X          57900000
```

- This instruction makes an external reference to BUFFER.
- the instruction is assembled using extended format with an address of 0.
- The x bit is set to 1 to indicate indexed addressing

190 0028 MAXLEN WORD BUFEND-BUFFER 000000

- value of data word to be generated is specified by an expression involving 2 external references.BUFFER and BUFFEND
- Assembler stores this value as 0
- when the program is loaded the loader will add to this data area the address of BUFFEND and subtract from it the address of BUFFER which result in desired value.

How to Handle External References

190 0028 MAXLEN WORD BUFEND-BUFFER 000000

- There are two external references in the expression, BUFEND and BUFFER since they are defined in another CS so values are unknown at assembly time
- The assembler inserts a value of zero
- passes information to the loader
- Add to this data area the address of BUFEND
- Subtract from this data area the address of BUFFER

107 1000 MAXLEN EQU BUFEND-BUFFER

On line 107, BUFEND and BUFFER are defined in the same control section and the expression can be calculated immediately.

- Any attempt to refer to a symbol in another CS is error unless it is identified as EXTREF
- Assembler must allow same symbol to be used in different CS
- **190 0028 MAXLEN WORD BUFEND-BUFFER 000000**
- **107 1000 MAXLEN EQU BUFEND-BUFFER**
- **reference to MAXLEN in CS COPY would use the definition in line 107**
- **Reference to MAXLEN in RDREC would use definition on line 190**
- **Assembler leaves room in the object code for values of external symbol**
- **Assembler include information in the object program for loader to insert proper value when required**

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 (hexadecimal)
  - Col.14-73 Repeat information in Col. 2-13 for other external symbols
- **Refer record**
  - Col. 1 D
  - Col. 2-7 Name of external symbol referred to in this control section
  - Col. 8-73 Name of other external reference symbols

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
- Gives information about external symbols that are defined in this control section
- symbols named by EXTDEF
- Refer record
- List symbols that are used as external reference by control section
- symbols named by EXTREF

Modification Record

- **Modification record(revised)**
  - Col. 1 M
  - Col. 2-7 Starting address of the field to be modified (hexadecimal)
  - Col. 8-9 Length of the field to be modified, in half-bytes (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.

- Modification record
  - M
  - Starting address of the field to be modified, relative to the beginning of the control section (Hex).
  - Length of the field to be modified, in *half-bytes*.
  - Modification flag (+ or -).
  - External symbol.

```
M^000004^05+RDREC
M^000028^06+BUFEND
M^000028^06-BUFFER
```

to

```
M00000705
M00001405
M00002705
```

```
M00000705+COPY
M00001405+COPY
M00002705+COPY
```

- Use Figure 2.8 for program relocation.

```
H^COPY  ^000000^001033
D^BUFFER^000033^BUFEND^001033^LENGTH^00002D
R^RDREC ^WRREC
T^000000^1D^1720^274B^100000^032023^290000^332007^4B^100000^3F2FEC^032016^0F2016
T^00001D^0D^010003^0F200A^4B^100000^3E2000
T^000030^03^454F46
M^000004^05^+RDREC
M^000011^05^+WRREC
M^000024^05^+WRREC
E^000000

H^RDREC ^000000^00002B
R^BUFFER^LENGTH^BUFEND
T^000000^1D^B410^B400^B440^77201^FE3201^B332FF^ADB201^5A0043^320095^790000^0E50
T^00001D^0E^3B2FE9^131000^004F0000^F1^000000
M^000018^05^+BUFFER
M^000021^05^+LENGTH
M^000028^06^+BUFEND
M^000028^06^-BUFFER
```

```
HWRREC 000000000001C
RLENGTHBUFFER
T00000001CB41077100000E32012332FFA53900000DF2008B8503B2FEE4F000005
M00000305+LENGTH
M00000D05+BUFFER
```

How Assembler to Handle Control Sections

- It processes each control section independently
- It establishes a separate LOCCTR (initialized to 0) for each control section
- It stores in SYMTAB the control section in which a symbol is defined
- It allows the same symbol to be used in different control sections
- It reports an error when attempting to refer to a symbol in another control section, unless the symbol is defined as an external reference
- References to unidentified external symbol are flagged as an error.
- It generate information in the object program for external references

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
  - legal Ex: BUFEND-BUFFER
  - illegal Ex: RDREC-COPY
-

External References in Expression

- **New restriction**
  - Both terms in each pair must be relative within the same control section
  - legal Ex: BUFEND-BUFFER
  - illegal Ex: RDREC-COPY

How to enforce this restriction

- When an expression involves external references, the assembler cannot determine whether or not the expression is legal at assembly time.
- Pairing of legal terms need knowledge of terms occuring in same CS which is unknown at assembly time.
- Assembler evaluate all terms,form expression,generate modification record ,
- The loader checks the expression for errors and finishes the evaluation.

Assembler Design Options

One-pass assemblers

Multi-pass assemblers

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

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

One-Pass Assemblers

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					

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

Load-and-go Assembler (Cont.)

- The actual starting address must be specified at assembly time
- Absolute program is used where actual address is known at assembly time and is assigned by system.

Producing Object Code

- **1 pass assembler producing object program as output is used where 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 Text 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**

PROGRAM				
Address	Label	Opcode	Operand	Object Code
	COPY	START	1000	
	EOF	BYTE	C'EOF'	
	THREE	WORD	3	
	ZERO	WORD	0	
	RETADR	RESW	1	
	LENGTH	RESW	1	
	BUFFER	RESB	4096	
	FIRST	STL	RETADR	
	CLOOP	JSUB	RDREC	
		LDA	LENGTH	
		COMP	ZERO	
		JEQ	ENDFILL	
		JSUB	WRREC	
		J	CLOOP	
	ENDFILL	LDA	EOF	
		STA	BUFFER	
		LDA	THREE	
		STA	LENGTH	
		JSUB	WRREC	
		LDL	RETADR	

SYMTAB	
Opearnd	Address

MEMORY			
1000		201A	
1001		201B	
1002		201C	
1003		201D	
1004		201E	
1005		201F	
1006		2020	
1007		2021	
1008		2022	
		2023	
200F		2024	
2010		2025	
2011		2026	
2012		2027	
2013		2028	
2014		2029	
2015		202A	
2016		201B	
2017		201C	
2018		201D	
2019		201E	
		201F	

PROGRAM

Address	Label	Opcode	Operand	Object Code
	COPY	START	1000	
1000	EOF	BYTE	C'EOF'	454F46
1003	THREE	WORD	3	000003
1006	ZERO	WORD	0	000000
1009	RETADR	RESW	1	
100C	LENGTH	RESW	1	
100F	BUFFER	RESB	4096	
200F	FIRST	STL	RETADR	141009
2012	CLOOP	JSUB	RDREC	48****
2015		LDA	LENGTH	00100C
2018		COMP	ZERO	281006
201B		JEQ	ENDFILL	30****
201E		JSUB	WRREC	48****
2021		J	CLOOP	3C2012
2024	ENDFILL	LDA	EOF	
		STA	BUFFER	
		LDA	THREE	
		STA	LENGTH	
		JSUB	WRREC	
		LDL	RETADR	

SYMTAB

Opearnd	Address	
EOF	1000	
THREE	1003	
ZERO	1006	
RETADR	1009	
LENGTH	100C	
BUFFER	100F	
FIRST	200F	
CLOOP	2012	
RDREC		→ 2013
ENDFILL	2024	→ 201C
WRREC		→ 201F

MEMORY

1000	45	201A	06
1001	4F	201B	30
1002	46	201C	**
1003	00	201D	**
1004	00	201E	48
1005	03	201F	**
1006	00	2020	**
1007	00	2021	3C
1008	00	2022	20
		2023	12
200F	14	2024	
2010	10	2025	
2011	09	2026	
2012	48	2027	
2013	**	2028	
2014	**	2029	
2015	00	202A	
2016	10	201B	
2017	0C	201C	
2018	28	201D	
2019	10	201E	
		201F	

PROGRAM

Address	Label	Opcode	Operand	Object Code
	COPY	START	1000	
1000	EOF	BYTE	C'EOF'	454F46
1003	THREE	WORD	3	000003
1006	ZERO	WORD	0	000000
1009	RETADR	RESW	1	
100C	LENGTH	RESW	1	
100F	BUFFER	RESB	4096	
200F	FIRST	STL	RETADR	141009
2012	CLOOP	JSUB	RDREC	48****
2015		LDA	LENGTH	00100C
2018		COMP	ZERO	281006
201B		JEQ	ENDFILL	30****
201E		JSUB	WRREC	48****
2021		J	CLOOP	3C2012
2024	ENDFILL	LDA	EOF	001000
		STA	BUFFER	
		LDA	THREE	
		STA	LENGTH	
		JSUB	WRREC	
		LDL	RETADR	

SYMTAB

Opearnd	Address
EOF	1000
THREE	1003
ZERO	1006
RETADR	1009
LENGTH	100C
BUFFER	100F
FIRST	200F
CLOOP	2012
RDREC	→ 2013
ENDFILL	2024
WRREC	→ 201F

MEMORY

1000	45	201A	06
1001	4F	201B	30
1002	46	201C	20
1003	00	201D	24
1004	00	201E	48
1005	03	201F	**
1006	00	2020	**
1007	00	2021	3C
1008	00	2022	20
		2023	12
200F	14	2024	00
2010	10	2025	10
2011	09	2026	00
2012	48	2027	
2013	**	2028	
2014	**	2029	
2015	00	202A	
2016	10	201B	
2017	0C	201C	
2018	28	201D	
2019	10	201E	
		201F	

PROGRAM

Address	Label	Opcode	Operand	Object Code
	COPY	START	1000	
1000	EOF	BYTE	C'EOF'	454F46
1003	THREE	WORD	3	000003
1006	ZERO	WORD	0	000000
1009	RETADR	RESW	1	
100C	LENGTH	RESW	1	
100F	BUFFER	RESB	4096	
200F	FIRST	STL	RETADR	141009
2012	CLOOP	JSUB	RDREC	48****
2015		LDA	LENGTH	00100C
2018		COMP	ZERO	281006
201B		JEQ	ENDFILL	30****
201E		JSUB	WRREC	48****
2021		J	CLOOP	3C2012
2024	ENDFILL	LDA	EOF	001000
2027		STA	BUFFER	0C100F
202A		LDA	THREE	001003
202D		STA	LENGTH	0C100C
2030		JSUB	WRREC	48
		LDL	RETADR	

SYMTAB

Opearnd	Address
EOF	1000
THREE	1003
ZERO	1006
RETADR	1009
LENGTH	100C
BUFFER	100F
FIRST	200F
CLOOP	2012
RDREC	→ 2013
ENDFILL	2024
WRREC	→ 201F

MEMORY

1000	45
1001	4F
1002	46
1003	00
1004	00
1005	03
1006	00
1007	00
1008	00
200F	14
2010	10
2011	09
2012	48
2013	**
2014	**
2015	00
2016	10
2017	0C
2018	28
2019	10

201A	06
201B	30
201C	20
201D	24
201E	48
201F	**
2020	**
2021	3C
2022	20
2023	12
2024	00
2025	10
2026	00
2027	0C
2028	10
2029	0F
202A	00
201B	10
201C	03
201D	0C
201E	10
201F	0C

2030	48
2031	20
2032	62
2033	08
2034	10
2035	09
2036	
2037	
2038	

Multi-Pass Assemblers

- **Restriction on EQU and ORG**
  - no forward reference, since symbols' value can't be defined during the first pass
  - restriction(symbols used on the right hand side be defined previously in the source program.
  - Normally applied to all assembler directives that define symbols
- **Example**
  - Use link list to keep track of whose value depend on an undefined symbol
- **Figure 2.21**

Need for a Multi-Pass Assemblers

- 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.

Need for a Multi-Pass Assemblers

- Make as many passes as are needed to process the definition of symbols
- Not necessary to make more than 2 passes over the entire program
- Portion of the program with forward reference in symbol definition are saved during Pass1 .
- Additonal passes done on these stored definitions
- Process is followed by normal Pass 2

- 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.

Table also indicates which symbols are depended on the values of others for evaluation

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

2 MAXLEN EQU BUFEND-BUFFER

3 PREVBT EQU BUFFER-1

4 BUFFER RESB 4096

5 BUFEND EQU *

```
A    EQU    B/2
B    EQU    C-D
E    EQU    D-1
D    RESB    4096
C    EQU    *
```


```
A    EQU    B/2
B    EQU    C-D
E    EQU    D-1
D    RESB    4096
C    EQU    *
```

A	&1	B/2	ϕ
B		*	→ A ϕ

A	&1	B/2	ϕ
B	&2	C-D	→ A ϕ
C		*	→ B ϕ
D		*	→ B ϕ
E	&1	D-1	ϕ

A	&1	B/2	ϕ
B	&2	C-D	
C		*	
D		*	
E	&1	D-1	ϕ

A	&1	B/2	ϕ
B	&2	C-D	
C		*	
D		*	
E	&1	D-1	ϕ

Suppose address of D RESB 4096 is 1034
so D will have value 1034

A	&1	B/2	ϕ
B	&1	C-1034	
C		*	
D		1034	ϕ
E		1034-1	ϕ

A	ϕ

B	ϕ

C contains the current value of Locctr=1034+1000(hex of 4096)=2034

A	&1	B/2	ϕ
B	&1	C-1034	
C		2034	
D		1034	ϕ
E		1033	ϕ

A	ϕ

B	ϕ

A	&1	B/2	ϕ
B		2034-1034	
C		2034	ϕ
D		1034	ϕ
E		1033	ϕ

A	ϕ

A	&1	B/2	ϕ
B		1000	
C		2034	ϕ
D		1034	ϕ
E		1033	ϕ

A	ϕ

A		1000/2	ϕ
B		1000	ϕ
C		2034	ϕ
D		1034	ϕ
E		1033	ϕ

```
A    EQU    B/2
B    EQU    C-D
E    EQU    D-1
D    RESB    4096
C    EQU    *
```

A		800	ϕ
B		1000	ϕ
C		2034	ϕ
D		1034	ϕ
E		1033	ϕ

Implementation Examples

Microsoft MASM Assembler

SEGMENT

- a collection segments, each segment is defined as belonging to a particular class, CODE, DATA, CONST, STACK
- registers: CS (code), SS (stack), DS (data), ES, FS, GS
- similar to program blocks in SIC
- ASSUME

```
e.g. ASSUME    ES:DATASEG2
e.g. MOVE      AX, DATA
     MOVE      ES,AX
```

similar to BASE in SIC