Chapter 2
Assemblers

Outline

- 2.1 Basic Assembler Functions
- 2.2 Machine-Dependent Assembler Features
- 2.3 Machine-Independent Assembler Features
- 2.4 Assembler Design Options
- 2.5 Implementation Examples

Introduction to Assemblers

- Fundamental functions
  - Translate mnemonic operation codes to their *machine language* equivalents
  - Assign *machine addresses* to *symbolic labels* used by the programmer
- The feature and design of an assembler depend
  - *Source language* it translate
  - The *machine language* it produce

2.1 Basic Assembler Functions

- ***Assembler***
  - A program that accepts an *assembly language program* as input and produces its *machine language equivalent* along with *information for the loader*

2.1 Basic Assembler Functions (Cont.)

- Constructions of assembly language program
  - ***Instruction***

    Label mnemonic operand

    - Operand
      - Direct addressing
        - E.g. LDA ZERO
      - Immediate addressing
        - E.g. LDA #0
      - Indexed addressing
        - E.g. STCH BUFFER, X
      - Indirect addressing
        - E.g J @RETADR

2.1 Basic Assembler Functions (Cont.)

- Constructions of assembly language program (Cont.)
  - ***Data***

    Label BYTE value

    Label WORD value

    Label RESB value

    Label RESW value

    - Label: name of operand
    - value: integer, character
    - E.g. EOF BYTE C'EOF'
    - E.g. FIVE WORD 5

Assembler Directives

- **Pseudo-instructions**
  - Not translated into machine instructions
  - Provide instructions to the assembler itself
- Basic assembler directives
  - **START:** specify *name* and *starting address of the program*
  - **END:** specify *end of program* and (option) *the first executable instruction in the program*
    - If not specified, use the address of the first executable instruction
  - **BYTE:** direct the assembler to *generate constants*
  - **WORD**
  - **RESB:** : instruct the assembler to *reserve memory location* without generating data values
  - **RESW**

Example of a SIC Assembler Language Program (Fig 2.1)

- Goal:
  - Reads records from input device (code F1)
  - Copies them to output device (code 05)
  - Loop until end of the file is detected
    - Write EOF on the output device
    - Terminate by executing an RSUB instruction to return to the operating system
      - Assume this program is called by OS using JSUB

Example of a SIC Assembler Language Program (Fig 2.1) (Cont.)

- Data transfer (RD, WD) method for each record
  - A buffer is used to store record
    - Buffering is necessary for different I/O rate devices
  - The end of each record is marked with a NULL character (00_{16})
    - The end of file is indicated by a *zero-length record*
- Subroutines (JSUB, RSUB) are used
  - RDREC, WRREC
  - Save *link register* first before nested jump

Example of a SIC Assembler Language Program (Fig 2.2)

- Show the generated *object code* for each statement in Fig. 2.1
- ***Loc*** column shows the *machine address* for each part of the assembled program
  - Assume program starts at address 1000
  - All instructions, data, or reserved storage are *sequential arranged* according to their order in source program.
  - A ***location counter*** is used to keep track the address changing

Example of a SIC Assembler Language Program (Fig 2.1,2.2)

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	

Example of a SIC Assembler Language Program (Fig 2.1,2.2) (Cont.)

```
                .
                .       SUBROUTINE TO READ RECORD INTO BUFFER
                .
2039    RDREC   LDX     ZERO            041030
203C            LDA     ZERO            001030
203F    RLOOP   TD      INPUT           E0205D
2042            JEQ     RLOOP           30203F
2045            RD      INPUT           D8205D
2048            COMP    ZERO            281030
204B            JEQ     EXIT            302057
204E            STCH    BUFFER,X        549039
2051            TIX     MAXLEN          2C205E
2054            JLT     RLOOP           38203F
2057    EXIT    STX     LENGTH          101036
205A            RSUB                    4C0000
205D    INPUT   BYTE    X'F1'           F1
205E    MAXLEN  WORD    4096            001000
```

Example of a SIC Assembler Language Program (Fig 2.1,2.2) (Cont.)

```
                 .               SUBROUTINE TO WRITE RECORD FROM BUFFER
                 .
                 .
        2061     WRREC     LDX       ZERO                041030
        2064     WLOOP     TD        OUTPUT              E02079
        2067               JEQ       WLOOP               302064
        206A               LDCH      BUFFER,X            509039
        206D               WD        OUTPUT              DC2079
        2070               TIX       LENGTH              2C1036
        2073               JLT       WLOOP               382064
        2076               RSUB                          4C0000
        2079     OUTPUT    BYTE      X'05'               05
                           END       FIRST
```

Figure 2.2 Program from Fig. 2.1 with object code.

Functions of a Basic Assembler

- Convert *mnemonic operation codes* to their *machine language equivalents*
  - E.g. STL -> 14 (line 10)
- Convert *symbolic operands* to their equivalent *machine addresses*
  - E.g. RETADR -> 1033 (line 10)
- Build the machine instructions in the proper format
- Convert the *data constants* to *internal machine representations*
  - E.g. EOF -> 454F46 (line 80)
- Write the *object program* and the *assembly listing*

Functions of a Basic Assembler (Cont.)

- All of above functions can be accomplished by *sequential processing* of the source program
  - *Except number 2 in processing symbolic operands*
- Example
  - **10 STL RETADR**
    - *RETADR* is not yet defined when we encounter *STL* instruction
    - Called *forward reference*

Symbolic Operands (Renew)

- We're not likely to write *memory addresses* directly in our code.
  - Instead, we will define *variable names*.
- Other examples of symbolic operands
  - Labels (for jump instructions)
  - Subroutines
  - Constants

Address Translation Problem

- *Forward reference*
  - A reference to a label that is defined later in the program
    - We will be unable to process this statement
- As a result, most assemblers make 2 passes over the source program
  - ***1st pass***: scan *label definitions* and *assign addresses*
  - ***2nd pass***: actual translation (object code)

Functions of Two Pass Assembler

- **Pass 1 - define symbols (assign addresses)**
  - Assign addresses to all statements in the program
  - Save the values assigned to all labels for use in Pass 2
  - Process some assembler directives
- **Pass 2 - assemble instructions and generate object program**
  - Assemble instructions
  - Generate data values defined by BYTE, WORD, etc.
  - Process the assembler directives not done in Pass 1
  - Write the object program and the assembly listing

Object Program

- Finally, assembler must write the generated object code to some output device
  - Called *object program*
  - Will be later loaded into memory for execution

Object Program (Cont.)

- Contains 3 types of records:
  - **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 (hex) (2 columns per byte)

  - **End record**

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

Object Program for Fig 2.2 (Fig 2.3)

Program name,Starting address (hex),Length of object program in bytes (hex)

```
H^COPY  ^001000^00107A
T^001000^1E^141033^482039^001036^281030^301015^482061^3C1003^00102A^0C1039^00102D
T^00101E^15^0C1036^482061^081033^4C0000^454F46^000003^000000
T^002039^1E^041030^001030^E0205D^30203F^D8205D^281030^302057^549039^2C205E^38203F
T^002057^1C^101036^4C0000^F1^001000^041030^E02079^302064^509039^DC2079^2C1036
T^002073^07^382064^4C0000^05
E^001000
```

2.3 Object program corres

Address of first executable instruction (hex)

Starting address (hex),Length of object code in this record (hex),Object code (hex)

2.1.2 Assembler Algorithm and Data Structures

- Algorithm
  - Two-pass assembler

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

Internal Data Structures

- **OPTAB (operation code table)**
  - Content
    - Menmonic machine code and its machine language equivalent
    - May also include instruction format, length etc.
  - Usage
    - Pass 1: used to loop up and validate operation codes in the source program
    - Pass 2: used to translate the operation codes to machine language
  - Characteristics
    - Static table, predefined when the assembler is written
  - Implementation
    - Array or hash table with mnemonic operation code as the key (preferred)
  - Ref. Appendix A

Internal Data Structures (Cont.)

- **SYMTAB (symbol table)**
  - Content
    - Label name and its value (address)
    - May also include flag (type, length) etc.
  - Usage
    - Pass 1: labels are entered into SYMTAB with their address (from LOCCTR) as they are encountered in the source program
    - Pass 2: symbols used as operands are looked up in SYMTAB to obtain the address to be inserted in the assembled instruction
  - Characteristic
    - Dynamic table (insert, delete, search)
  - Implementation
    - Hash table for efficiency of *insertion* and *retrieval*

Internal Data Structures (Cont.)

- **Location Counter**
  - A variable used to help in *assignment of addresses*
  - Initialized to the beginning address specified in the START statement
  - Counted in bytes

Algorithm for 2 Pass Assembler (Fig 2.4)

- Figure 2.4 (a): algorithm for pass 1 of assembler

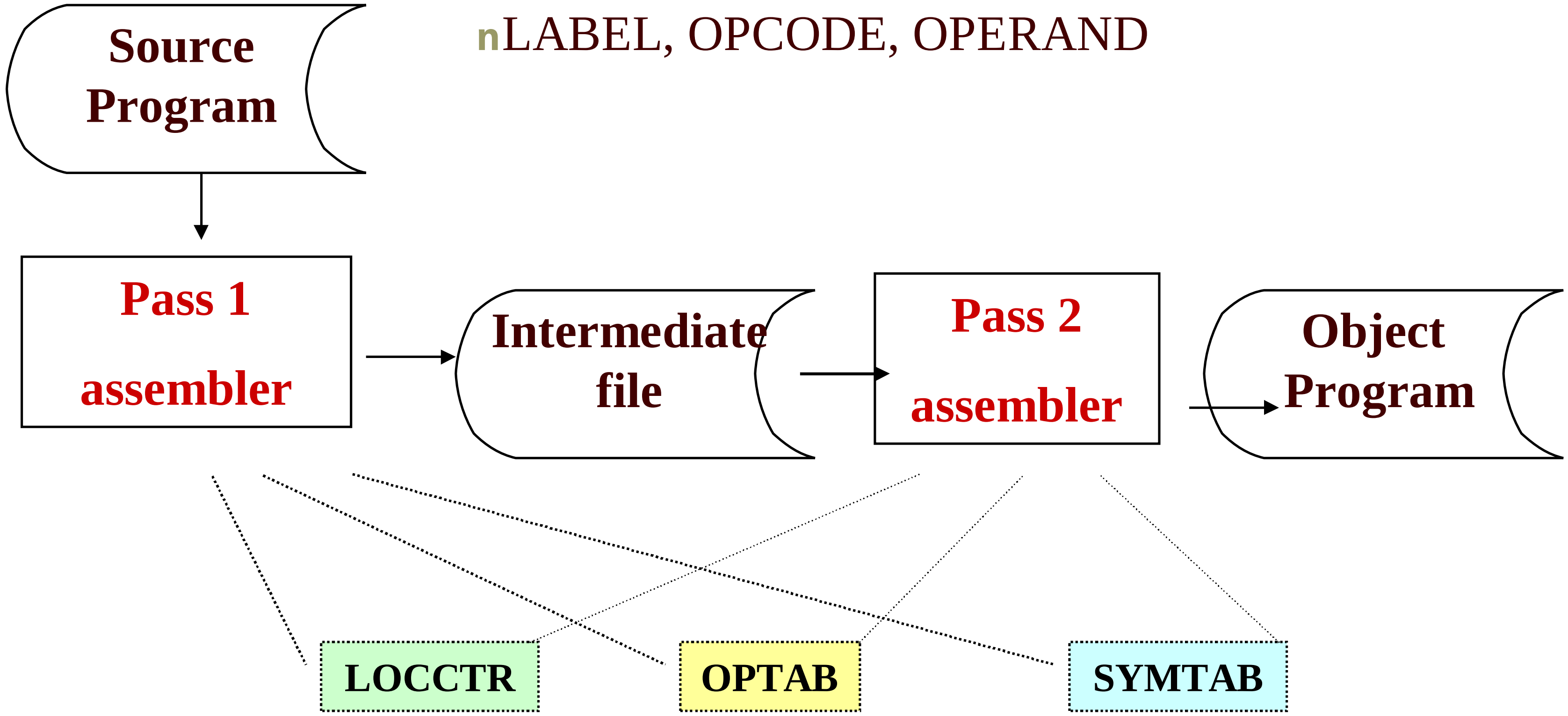
- Figure 2.4 (b): algorithm for pass 2 of assembler

Algorithm for 2 Pass Assembler (Fig 2.4)

- Both pass1 and pass 2 need to read the source program.
  - However, pass 2 needs more information
    - Location counter value, error flags
- **Intermediate file**
  - Contains each source statement with its assigned address, error indicators, etc
  - Used as the input to Pass 2

Intermediate File

- LABEL, OPCODE, OPERAND



Algorithm for Pass 1 of Assembler (Fig 2.4a)

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}
```

Figure 2.4(a) Algorithm for Pass 1 of assembler.

Algorithm for Pass 2 of Assembler (Fig 2.4b)

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}
```

Figure 2.4(b) Algorithm for Pass 2 of 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
- Assembler design Options
  - one-pass assemblers
  - multi-pass assemblers

2.2 Machine Dependent Assembler Features

- Machine Dependent Assembler Features
  - SIC/XE
  - Instruction formats and addressing modes
  - Program relocation

SIC/XE Assembler

- Previous, we know how to implement the 2-pass SIC assembler.

- What's new for SIC/XE?
  - More addressing modes.
  - Program Relocation.

SIC/XE Assembler (Cont.)

- SIC/XE
  - Immediate addressing: op #c
  - Indirect addressing: op @m
  - PC-relative or Base-relative addressing: op m
    - The assembler directive **BASE** is used with base-relative addressing
    - If displacements are too large to fit into a 3-byte instruction, then 4-byte extended format is used
  - Extended format: +op m
  - Indexed addressing: op m, **x**
  - Register-to-register instructions
  - Large memory
    - Support multiprogramming and need *program reallocation* capability

Example of a SIC/XE Program (Fig 2.5)

- Improve the execution speed
  - Register-to-register instructions
  - Immediate addressing: op #c
    - Operand is already present as part of the instruction
  - Indirect addressing: op @m
    - Often avoid the need of another instruction

Example of a SIC/XE Program (Fig 2.5,2.6)

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	

Example of a SIC/XE Program (Fig 2.5,2.6) (Cont.)

```
110                  .
115                  .        SUBROUTINE TO READ RECORD INTO BUFFER
120                  .
125     1036    RDREC      CLEAR     X                 B410
130     1038               CLEAR     A                 B400
132     103A               CLEAR     S                 B440
133     103C              +LDT       #4096             75101000
135     1040    RLOOP      TD        INPUT             E32019
140     1043               JEQ       RLOOP             332FFA
145     1046               RD        INPUT             DB2013
150     1049               COMPR     A,S               A004
155     104B               JEQ       EXIT              332008
160     104E               STCH      BUFFER,X          57C003
165     1051               TIXR      T                 B850
170     1053               JLT       RLOOP             3B2FEA
175     1056    EXIT       STX       LENGTH            134000
180     1059               RSUB                        4F0000
185     105C    INPUT      BYTE      X'F1'             F1
```

Example of a SIC/XE Program (Fig 2.5,2.6) (Cont.)

195		.			
200		.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205		.			
210	105D	WRREC	CLEAR	X	B410
212	105F		LDT	LENGTH	774000
215	1062	WLOOP	TD	OUTPUT	E32011
220	1065		JEQ	WLOOP	332FFA
225	1068		LDCH	BUFFER,X	53C003
230	106B		WD	OUTPUT	DF2008
235	106E		TIXR	T	B850
240	1070		JLT	WLOOP	3B2FEF
245	1073		RSUB		4F0000
250	1076	OUTPUT	BYTE	X'05'	05
255			END	FIRST	

Figure 2.6 Program from Fig. 2.5 with object code.

2.2.1 Instruction Formats and Addressing Modes

- START now specifies a beginning program address of 0
  - Indicate a *relocatable program*
- Register translation
  - For example: *COMPR A, S => A004*
  - Must keep the register name (A, X, L, B, S, T, F, PC, SW) and their values (0,1, 2, 3, 4, 5, 6, 8, 9)
    - Keep in SYMTAB

Address Translation

- Most register-to-memory instructions are assembled using *PC relative* or *base relative* addressing
  - Assembler must calculate a *displacement* as part of the object instruction
  - If displacement can be fit into 12-bit field, format 3 is used.
  - Format 3: 12-bit address field
    - Base-relative: 0~4095
    - PC-relative: -2048~2047
  - Assembler attempts to translate using PC-relative first, then base-relative
    - If displacement in PC-relative is out of range, then try base-relative

Address Translation (Cont.)

- If displacement can not be fit into 12-bit field in the object instruction, format 4 must be used.
  - Format 4: 20-bit address field
  - No displacement need to be calculated.
    - 20-bit is large enough to contain the full memory address
  - Programmer must specify extended format: +op   m
  - For example: *+JSUB   RDREC => 4B101036*
    - *LOC(RDREC) = 1036, get it from SYMTAB*

PC-Relative Addressing Modes

- *10 0000 FIRST STL RETADR 17202D*
  - Displacement= RETADR – (PC) = 30-**3** = 2D
  - Opcode (6 bits) $=14_{16}=00010100_2$
  - nixbpe=110010
    - n=1, i = 1: indicate neither *indirect* nor *immediate* addressing
    - p = 1: indicate *PC-relative* addressing

OPCODE	n	i	x	b	p	e	Address
0001 01	1	1	0	0	1	0	$(02D)_{16}$

Object Code = 17202D

PC-Relative Addressing Modes (Cont.)

- 40 0017 J CLOOP 3F2FEC
  - Displacement= CLOOP - (PC) = 6 - **1A** = -14 = FEC (2's complement for negative number)
  - Opcode=$3C_{16}$ = 00111100_2
  - nixbpe=110010

OPCODE	n	i	x	b	p	e	Address
0011 11	1	1	0	0	1	0	$(FEC)_{16}$

Object Code = 3F2FEC

Base-Relative Addressing Modes

- Base register is under the control of the programmer
  - Programmer use assembler directive ***BASE*** to specify which value to be assigned to base register (B)
  - Assembler directive ***NOBASE***: inform the assembler that the contents of base register no longer be used for addressing
  - ***BASE*** and ***NOBASE*** produce no executable code

Base-Relative Addressing Modes (Cont.)

- *12 LDB #LENGTH*
- *13 BASE LENGTH ;no object code*
- *160 104E STCH BUFFER, X 57C003*
  - Displacement= BUFFER – (B) = 0036 – 0033(=LOC(LENGTH)) = 3
  - Opcode=54
  - nixbpe=111100
    - n=1, i = 1: indicate neither *indirect* nor *immediate* addressing
    - x = 1: *indexed* addressing
    - b = 1: *base-relative* addressing

OPCODE	n	i	x	b	p	e	Address
0101 01	1	1	1	1	0	0	$(003)_{16}$

Object Code = 57C003

Address Translation

- Assembler attempts to translate using *PC-relative* first, then *base-relative*
  - e.g. 175 1053 STX LENGTH 134000
    - Try PC-relative first
      - Displacement= LENGTH - (PC) = 0033 - 1056 = -1026 (hex)
    - Try base-relative next
      - displacement= LENGTH – (B) = 0033 – 0033 =0
      - Opcode=10
      - nixbpe=110100
        - n=1, i = 1: indicate neither *indirect* nor *immediate* addressing
        - b = 1: *base-relative* addressing

Immediate Address Translation

- Convert the *immediate* operand to its internal representation and insert it into the instruction
- 55 0020 LDA #3 010003
  - Opcode=00
  - nixbpe=010000
    - i = 1: *immediate addressing*

OPCODE	n	i	x	b	p	e	Address
0000 00	0	1	0	0	0	0	$(003)_{16}$

Object Code = 010003

Immediate Address Translation (Cont.)

- 133 103C +LDT #4096 75101000
  - Opcode=74
  - nixbpe=010001
    - i = 1: *immediate addressing*
    - e = 1: *extended instruction format* since 4096 is too large to fit into the 12-bit displacement field

OPCODE	n	i	x	b	p	e	Address
0111 01	0	1	0	0	0	1	$(01000)_{16}$

Object Code = 75101000

Immediate Address Translation (Cont.)

- 12 0003 LDB #LENGTH 69202D
  - The immediate operand is the symbol LENGTH
    - The address of LENGTH is loaded into register B
  - Displacement=LENGTH – (PC) = 0033 – 0006 = 02D
  - Opcode=68_{16} = 01101000_2
  - nixbpe=010010
    - Combined *PC relative* (p=1) with *immediate addressing* (i=1)

OPCODE	n	i	x	b	p	e	Address
0110 10	0	1	0	0	1	0	$(02D)_{16}$

Immediate Address Translation (Cont.)

- 55 0020 LDA #3   010003
  - Opcode = 00_{16} = 00000000_2
  - nixbpe=010000
    - i = 1: immediate addressing

OPCODE	n	i	x	b	P	e	Address
0110 10	0	1	0	0		0	$(02D)_{16}$

Indirect Address Translation

- Indirect addressing
  - The contents stored at the location represent the *address* of the operand, not the operand itself
  - Target addressing is computed as usual (PC-relative or BASE-relative)
  - *n* bit is set to 1

Indirect Address Translation (Cont.)

- 70 002A J @RETADR 3E2003
  - Displacement= RETADR- (PC) = 0030 – 002D =3
  - Opcode= 3C
  - nixbpe=100010
    - n = 1: *indirect addressing*
    - p = 1: *PC-relative addressing*

OPCODE	n	i	x	b	p	e	Address
0011 11	1	0	0	0	1	0	$(003)_{16}$

Note

- Ref: ***Appendix A***

2.2.2 Program Relocation

- The larger main memory of SIC/XE
  - Several programs can be loaded and run at the same time.
  - This kind of sharing of the machine between programs is called ***multiprogramming***
- To take full advantage
  - Load programs into memory wherever there is room
  - Not specifying a fixed address at assembly time
  - Called ***program relocation***

2.2.2 Program Relocation (Cont.)

- *Absolute program* (or *absolute assembly*)
  - Program must be loaded at the address specified *at assembly time.*
  - E.g. Fig. 2.1

```
COPY   START   1000      <- program loading starting address 1000
FIRST  STL     RETADR
          :
          :
```

- e.g. 55 101B LDA THREE 00102D

  - What if the program is loaded to 2000

    e.g. 55 101B LDA THREE 00202D

    - Each absolute address should be modified

Example of Program Relocation (Fig 2.7)

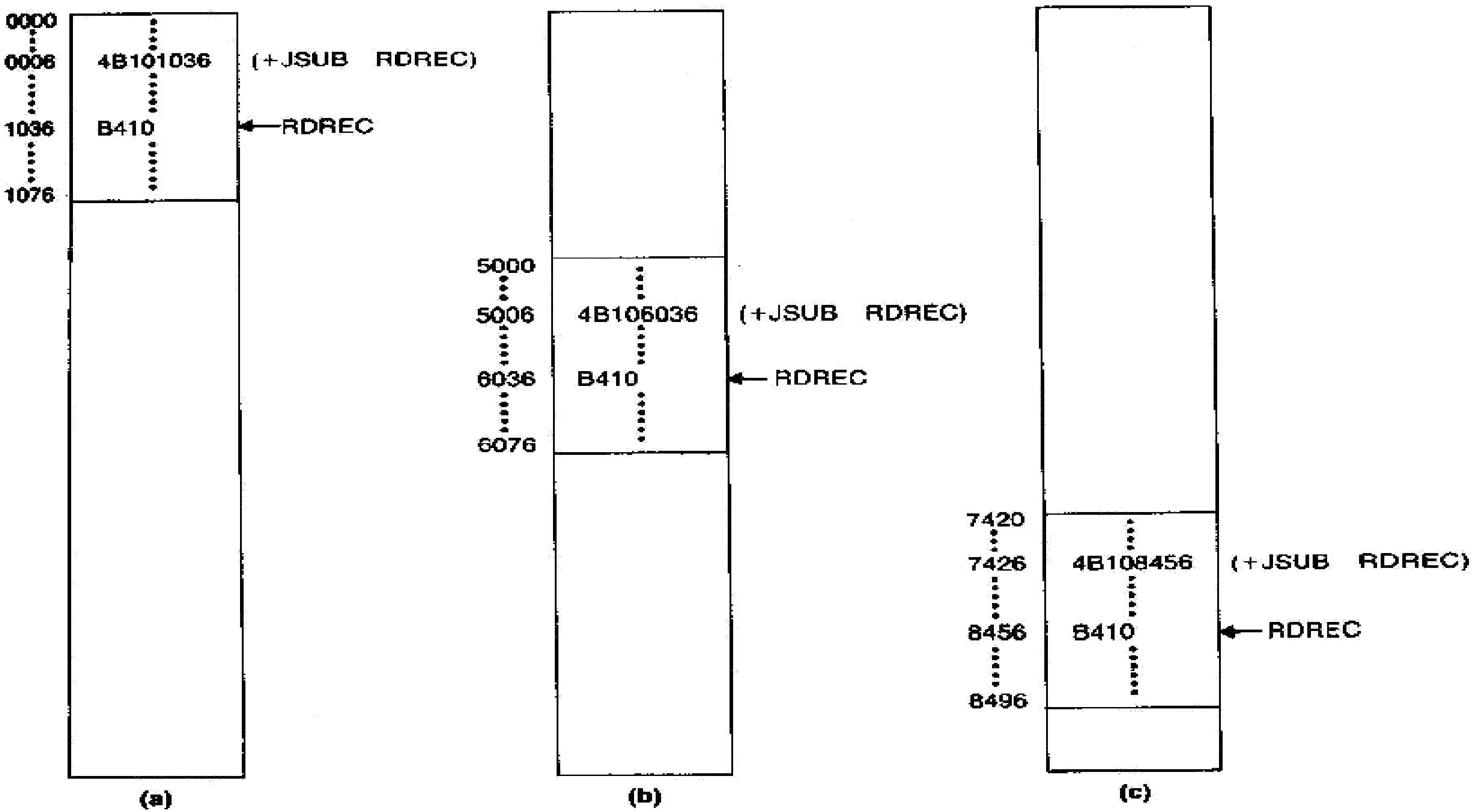


Figure 2.7 Examples of program relocation.

2.2.2 Program Relocation (Cont.)

- *Relocatable* program

```
COPY   START   0
FIRST  STL     RETADR
          :
          :
```

program loading starting address is determined *at load time*

  - An object program that contains the information necessary to perform *address modification* for relocation
  - The **assembler** must identify for the **loader** those parts of object program that need modification.
  - No instruction modification is needed for
    - Immediate addressing (not a memory address)
    - PC-relative, Base-relative addressing
  - The only parts of the program that require modification at load time are those that specify *direct addresses*
    - In SIC/XE, only found in extended format instructions

Instruction Format vs. Relocatable Loader

- In SIC/XE
  - Format 1, 2, 3
    - Not affect
  - Format 4
    - Should be modified
- In SIC
  - Format 3 with address field
    - Should be modified
    - SIC does not support PC-relative and base-relative addressing

Relocatable Program

- We use modification records that are added to the object files.

Pass the *address–modification* information to the relocatable loader

- *Modification record*
  - Col 1 M
  - Col 2-7 Starting location of the address field to be modified, relative to the beginning of the program (hex)
  - Col 8-9 length of the address field to be modified, in half-bytes
  - E.g M$_\wedge$000007$_\wedge$05

Beginning address of the program is to be added to a field that begins at addr ox000007 and is 5 bytes in length.

Object Program for Fig 2.6 (Fig 2.8)

```
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
```

Figure 2.8 Object program corresponding to Fig. 2.6.

2.3 Machine-Independent Assembler Features

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

2.3.1 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.
  - Such an operand is called a ***literal*** because the value is stated "literally" in the instruction.
  - A literal is identified with the prefix **=**
- Examples

45	001A	ENDFILLDA	=C'EOF'	032010
215	1062	WLOOPTD	=X'05'	E32011

Original Program (Fig. 2.6)

```
  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
```

Using Literal (Fig. 2.9)

```
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
```

Object Program Using 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
```

The same as before

Original Program (Fig. 2.6)

```
                .
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
```

Using Literal (Fig. 2.9)

```
        .
        .       SUBROUTINE TO WRITE RECORD FROM BUFFER
        .
WRREC   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 BUFFER
        WD      =X'05'          WRITE CHARACTER
        TIXR    T               LOOP UNTIL ALL CHARACTERS
        JLT     WLOOP             HAVE BEEN WRITTEN
        RSUB                    RETURN TO CALLER
        END     FIRST
```

Object Program Using Literal

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

The same as before

Object Program Using Literal (Fig 2.9 & 2.10)

Line	Loc	Source statement			Object code
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	
106	1036	BUFEND	EQU	*	
107	1000	MAXLEN	EQU	BUFEND-BUFFER	
110		.			

Object Program Using Literal (Fig 2.9 & 2.10) (Cont.)

```
110                 .
115                 .           SUBROUTINE TO READ RECORD INTO BUFFER
120                 .
125     1036        RDREC       CLEAR     X                  B410
130     1038                    CLEAR     A                  B400
132     103A                    CLEAR     S                  B440
133     103C                   +LDT       #MAXLEN            75101000
135     1040        RLOOP       TD        INPUT              E32019
140     1043                    JEQ       RLOOP              332FFA
145     1046                    RD        INPUT              DB2013
150     1049                    COMPR     A,S                A004
155     104B                    JEQ       EXIT               332008
160     104E                    STCH      BUFFER,X           57C003
165     1051                    TIXR      T                  B850
170     1053                    JLT       RLOOP              3B2FEA
175     1056        EXIT        STX       LENGTH             134000
180     1059                    RSUB                         4F0000
185     105C        INPUT       BYTE      X'F1'              F1
195
```

Object Program Using Literal (Fig 2.9 & 2.10) (Cont.)

195		.			
200		.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
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

Figure 2.10 Program from Fig. 2.9 with object code.

Literals vs. Immediate Operands

- Immediate Operands
  - The operand value is assembled as *part of the machine instruction*
  - e.g. 55 0020 LDA #3 010003
- Literals

Similar to define constant

  - The assembler generates the specified value as a constant *at some other memory location*
  - The effect of using a literal is exactly the same as if the programmer had *defined the constant* and used the *label* assigned to the constant as the instruction operand.
  - e.g. 45 001A ENDFIL LDA =C'EOF' 032010 (Fig. 2.9)
- Compare (Fig. 2.6)
  - e.g. 45 001A ENDFIL LDA EOF 032010
    80 002D EOF BYTE C'EOF' 454F46

Literal - Implementation

- *Literal pools*
  - All of the literal operands are gathered together into one or more *literal pools*
  - Normally, literal are placed at the end of the object program, i.e., following the END statement by the *assembler*
  - E.g., Fig. 2.10 (END statement)
    255 END FIRST
    1076 * =X'05' 05

Literal – Implementation (Cont.)

- In some case, *programmer* would like to place literals into a pool at some other location in the object program
  - Using assembler directive **LTORG** (see Fig. 2.10)
  - Create a literal pool that contains all of the literal operands used since the previous LTORG
  - e.g., 45 001A ENDFIL LDA =C'EOF' 032010 (Fig.2.10)

    93 **LTORG**

    002D * =C'EOF' 454F46
  - Reason: keep the literal operand close to the instruction referencing it
    - Allow *PC-relative addressing* possible

Literal - Implementation (Cont.)

- 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
    - Compare the character strings defining them
      - E.g., =X'05'
      - Easier to implement, but has potential problem (see next)
    - Or compare the generated data value
      - E.g.the literals =C'EOF' and =X'454F46' would specify identical operand value.
      - Better, but will increase the complexity of the assembler

Same symbols, only one address is assigned

Literal - Implementation (Cont.)

- Be careful when using literal whose value depends on their *location* in the program
- For example, a literal may represent the *current value* of the *location counter*
  - Denoted by *
  - "LDB =*" may result in different object code when it appear *in different location*
  - *Cannot consider as duplicate literals*

Basic Data Structure for Assembler to Handle Literal Operands

- *Data Structure: literal table - LITTAB*
  - Content
    - Literal name
    - The operand value and length
    - Address assigned to the operand
  - Implementation
    - Organized as a hash table, using literal name or value as key.

How the Assembler Handles Literals?

- Pass 1
  - Build LITTAB with literal name, operand value and length, (leaving the address unassigned).
  - Handle duplicate literals. (Ignore duplicate literals)
  - When encounter LTORG statement or end of the program, assign an address to each literal not yet assigned an address
    - Remember to update the PC value to assign each literal's address
- Pass 2
  - Search LITTAB for each literal operand encountered
  - Generate data values in the object program exactly as if they are generated by BYTE or WORD statements
  - Generate modification record for literals that represent an *address* in the program (e.g. a location counter value)

2.3.2 Symbol-Defining Statements

- ***Labels*** on instructions or data areas
  - The value of such a label is the *address* assigned to the statement on which it appears
- Defining symbols
  - All programmer to define symbols and specify their values
  - Format: symbol **EQU** value
    - Value can be *constant* or *expression involving constants and previously defined symbols*
  - Example
    - MAXLEN EQU 4096
    - +LDT #MAXLEN

2.3.2 Symbol-Defining Statements (Cont.)

- Usage:
  - Make the source program easier to understand
- How assembler handles it?
  - In pass 1: when the assembler encounters the EQU statement, it enters the symbol into SYMTAB for later reference.
  - In pass 2: assemble the instruction with the *value* of the symbol
    - Follow the previous approach

Examples of Symbol-Defining Statements

- E.g. +LDT #4096 (Fig 2.5)
  - MAXLEN EQU 4096
  - +LDT #MAXLEN
- E.g. define mnemonic names for registers
  - A EQU 0
  - X EQU 1
  - L EQU 2
  - …
- E.g. define names that reflect the logical function of the registers in the program
  - BASE EQU R1
  - COUNT EQU R2
  - INDEX EQU R3

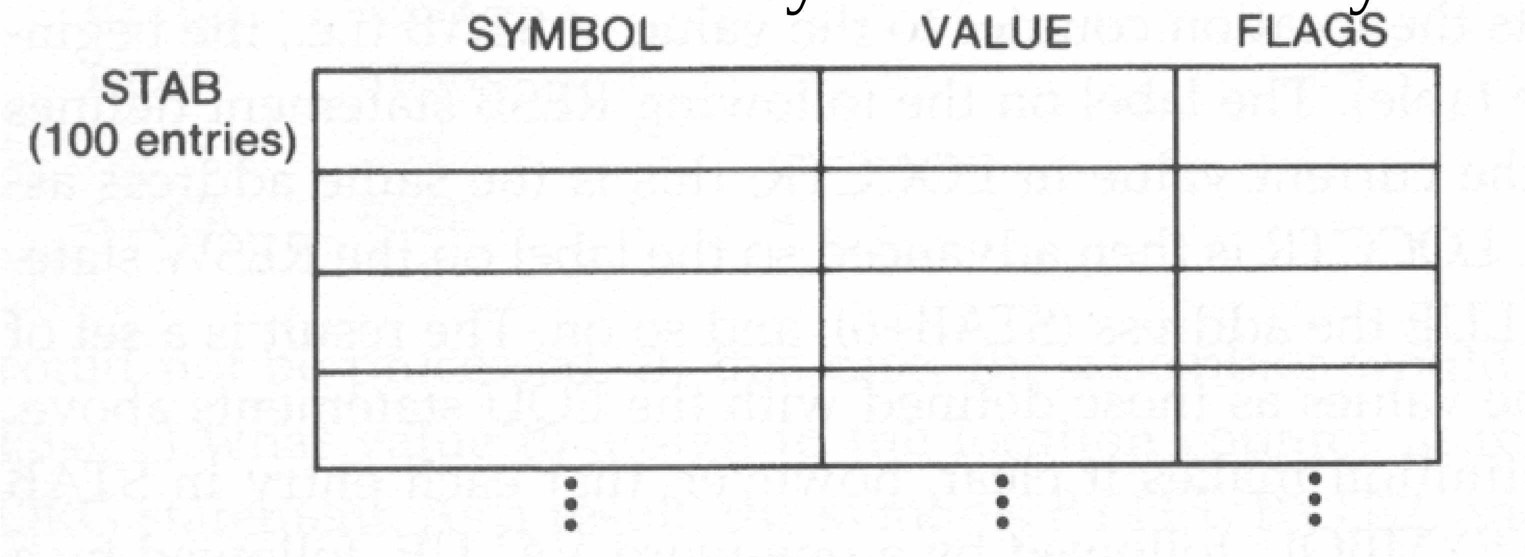
ORG

- ORG (origin)
  - Assembler directive: **ORG** value
    - Value can be *constant* or *expression involving constants and previously defined symbols*
  - Assembler resets the *location counter (LOCCTR)* to the specified value

LOCCTR control assignment of storage in the object program

Example of Using ORG

- Consider the following data structure
  - SYMBOL: 6 bytes
  - VALUE: 3 bytes (one word)
  - FLAGS: 2 bytes
- we want to refer to every field of each entry



ORG Example

- Using EQU statements

STAB	RESB	1100
SYMBOL	EQU	STAB
VALUE	EQU	STAB+6
FLAG	EQU	STAB+9

  - We can fetch the VALUE field by

    LDA VALUE,X

  - X = 0, 11, 22, … for each entry

Refer to entries in the table using indexed addressing

ORG Example (Cont.)

- Using ORG statements

STAB	RESB	1100
	ORG	STAB
SYMBOL	RESB	6
VALUE	RESW	1
FLAGS	RESB	2
	ORG	STAB+1100

Set the LOCCTR to STAB

Size of field more meaningful

Restore the LOCCTR to its previous value

  - This method of definition makes the structure more clear.
  - The last ORG is very important
    - Set program counter (LOCCTR) back to its previous value

Forward Reference

- All symbol-defining directives do ***not*** allow forward reference for 2-pass assembler
  - e.g., EQU, ORG…
  - All symbols used on the *right-hand side* of the statement must have been defined previously

E.g. (Cannot be assembled in 2-pass assm.)

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

Forward Reference (Cont.)

- E.g. (Cannot be assembled in 2-pass assm.)

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

2.3.3 Expressions

- Most assemblers allow the use of *expression* to replace symbol in the operand field.
  - Expression is evaluated by the assembler
  - Formed according to the rules using the operators +, -, *, /
    - Division is usually defined to produce an integer result
    - Individual terms can be
      - Constants
      - User-defined symbols
      - Special terms: e.g., * (= current value of location counter)

2.3.3 Expressions (Cont.)

- Review
  - Values in the object program are
    - *relative* to the beginning of the program or
    - *absolute* (independent of program location)
  - For example
    - Constants: absolute
    - Labels: relative

2.3.3 Expressions (Cont.)

- **Expressions can also be classified as *absolute expressions* or *relative expressions***
  - E.g. (Fig 2.9)

    ```
    107    MAXLEN          EQU   BUFEND-BUFFER
    ```

    - Both BUFEND and BUFFER are *relative terms*, representing addresses within the program
    - However the expression BUFEND-BUFFER represents an *absolute value: the difference between the two addresses*
  - When relative terms are paired with opposite signs
    - The dependency on the program starting address is canceled out
    - The result is an ***absolute value***

2.3.3 Expressions (Cont.)

- Absolute expressions
  - An expression that contains only absolute terms
  - An expression that contain relative terms but *in pairs* and the terms in each such pair have *opposite* signs
- Relative expressions
  - All of the relative terms *except one* can be paired and the remaining *unpaired relative terms* must have a *positive sign*
- ***No** relative terms* can enter into a multiplication or division operation no matter in absolute or relative expression

2.3.3 Expressions (Cont.)

- Errors: **(represent neither absolute values nor locations within the program)**
  - BUFEND+BUFFER // not opposite terms
  - 100-BUFFER // not in pair
  - 3*BUFFER // multiplication

2.3.3 Expressions (Cont.)

- Assemblers should determine the type of an expression
  - Keep track of the *types* of all symbols defined in the program in the symbol table.
  - Generate *Modification records* in the object program for relative values.

SYMTAB for Fig. 2.10

Symbol	Type	Value
RETADR	R	30
BUFFER	R	36
BUFEND	R	1036
MAXLEN	A	1000

2.3.4 Program Blocks

- Previously, main program, subroutines, and data area are treated as a unit and are assembled at the same time.
  - Although the source program logically contains subroutines, data area, etc, they were assembled into a single block of object code
  - To improve memory utilization, main program, subroutines, and data blocks may be allocated in separate areas.
- Two approaches to provide such a flexibility:
  - Program blocks
    - Segments of code that are rearranged within a single object program unit
  - Control sections
    - Segments of code that are translated into independent object program units

2.3.4 Program Blocks

- *Solution 1: Program blocks*
  - Refer to segments of code that are rearranged within a single object program unit
  - **Assembler directive: USE blockname**
    - Indicates which portions of the source program belong to which blocks.
  - Codes or data with same block name will allocate together
  - 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.

2.3.4 Program Blocks (Cont.)

- E.g: Figure 2.11
  - Three blocks
    - First: unnamed, i.e., default block
      - Line 5~ Line 70 + Line 123 ~ Line 180 + Line 208 ~ Line 245
    - Second: CDATA
      - Line 92 ~ Line 100 + Line 183 ~ Line 185 + Line 252 ~ Line 255
    - Third: CBLKS
      - Line 105 ~ Line 107
  - 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.

Program with Multiple Program Blocks (Fig 2.11 & 2.12)

Line	Loc/Block		Source statement			Object code
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	

Program with Multiple Program Blocks (Fig 2.11 & 2.12) (Cont.)

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
195			.			

Program with Multiple Program Blocks (Fig 2.11 & 2.12)

```
195                              .
200                              .        SUBROUTINE TO WRITE RECORD FROM BUFFER
205                              .
208     004D   0                          USE
210     004D   0    WRREC                 CLEAR     X                      B410
212     004F   0                          LDT       LENGTH                 772017
215     0052   0    WLOOP                 TD        =X'05'                 E3201B
220     0055   0                          JEQ       WLOOP                  332FFA
225     0058   0                          LDCH      BUFFER,X               53A016
230     005B   0                          WD        =X'05'                 DF2012
235     005E   0                          TIXR      T                      B850
240     0060   0                          JLT       WLOOP                  3B2FEF
245     0063   0                          RSUB                             4F0000
252     0007   1                          USE       CDATA
253                                       LTORG
        0007   1    *                     =C'EOF'                          454F46
        000A   1    *                     =X'05'                           05
255                                       END       FIRST
```

Figure 2.12 Program from Fig. 2.11 with object code.

Basic Data Structure for Assembler to Handle Program Blocks

- *Block name table*
  - Block name, block number, address, length

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

How the Assembler Handles Program Blocks?

- Pass 1
  - Maintaining ***separate** location counter* for each program block
  - Each label is assigned an address that is relative to ***the start of the block*** that contains it
  - When labels are entered into SYMTAB, the ***block name*** or ***number*** is stored along with the assigned relative addresses.
  - 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

How the Assembler Handles Program Blocks? (Cont.)

- 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 the start of its block
    - The assembler needs the address for each symbol *relative to the start of the object program*, not the start of an individual program block

Table for Program Blocks

- At the end of Pass 1 in Fig 2.11:

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

Example of Address Calculation

- Each source line is given a *relative address assigned* and a *block number*
  - ***Loc/Block*** Column in Fig. 2.11
- For an *absolute symbol* (whose value is not relative to the start of any program block), there is no block number
  - E.g. 107 1000 MAXLEN EQU BUFEND-BUFFER
- Example: calculation of address in Pass 2
  - 20 0006 0 LDA LENGTH 032060

LENGTH = (block 1 starting address)+0003 = 0066+0003= 0069

LOCCTR = (block 0 starting address)+0009 = 0009

PC-relative: Displacement = 0069 - (LOCCTR) = 0069-0009=0060

2.3.4 Program Blocks (Cont.)

- Program blocks reduce addressing problem:
  - No needs for extended format instructions (lines 15, 35, 65)
    - The larger buffer is moved to the end of the object program
  - No needs for base relative addressing (line 13, 14)
    - The larger buffer is moved to the end of the object program
  - LTORG is used to make sure the literals are placed ahead of any large data areas (line 253)
    - Prevent literal definition from its usage too far

2.3.4 Program Blocks (Cont.)

- Object code
  - It is not necessary to physically rearrange the generated code in the object program to place the pieces of each program block together.
  - Loader will load the object code from each record at the *indicated addresses*.
- For example (Fig. 2.13)
  - The first two Text records are generated from line 5~70
  - When the USE statement is recognized
    - Assembler writes out the current Text record, even if there still room left in it
    - Begin a new Text record for the new program block

Object Program Corresponding to Fig. 2.11 (Fig. 2.13)

```
H^COPY  ^000000^001071
T^000000^1E^172063^4B2021^032060^290000^332006^4B203B^3F2FEE^032055^0F2056^010003
T^00001E^09^0F2048^4B2029^3E203F
T^000027^1D^B410^B400^B440^75101000^E32038^332FFA^DB2032^A004^332008^57A02F^B850
T^000044^09^3B2FEA^13201F^4F0000
T^00006C^01^F1
T^00004D^19^B410^772017^E3201B^332FFA^53A016^DF2012^B850^3B2FEF^4F0000
T^00006D^04^454F46^05
E^000000
```

Figure 2.13 Object program corresponding to Fig. 2.11.

Program blocks for the Assembly and Loading Processes (Fig. 2.14)

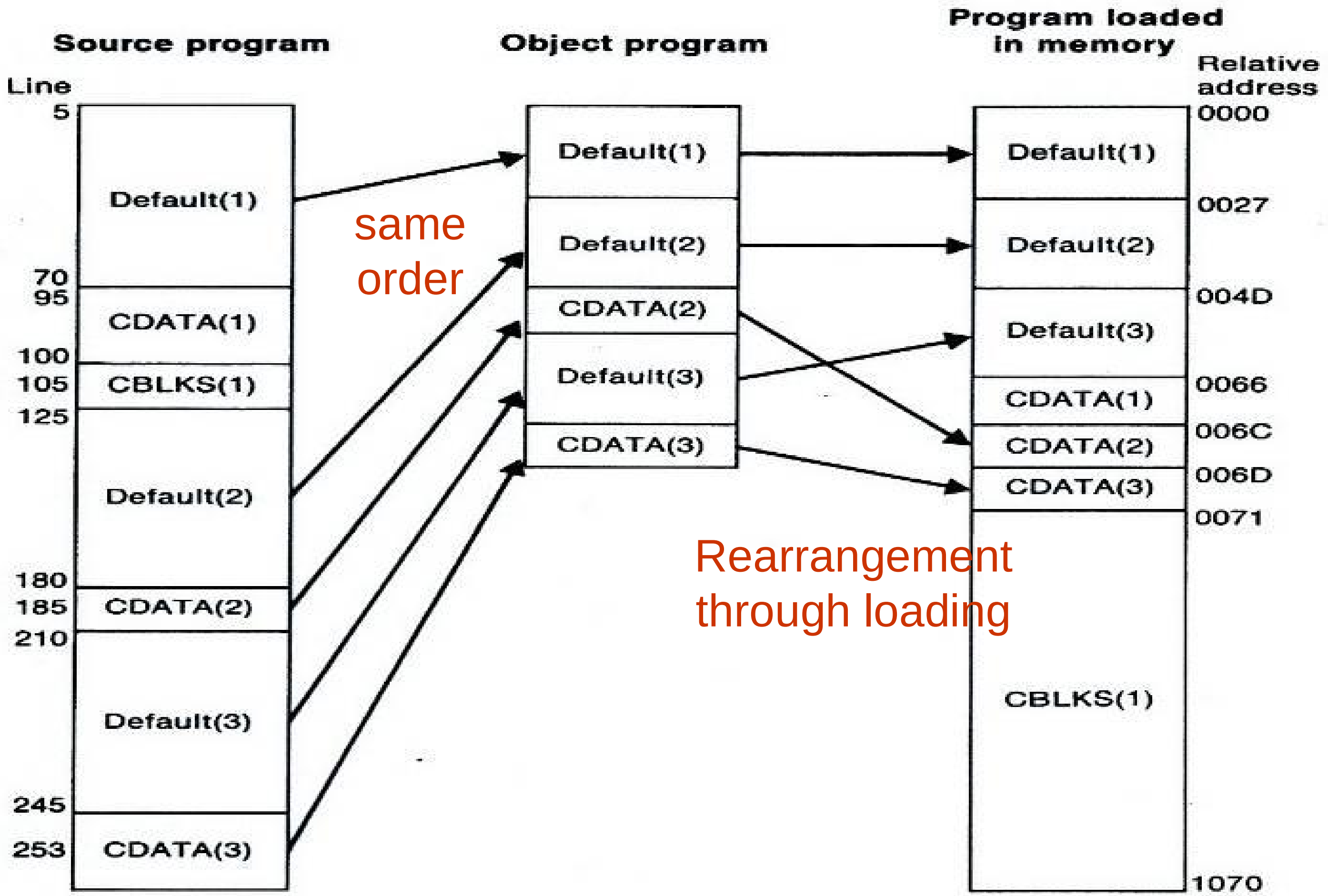


Figure 2.14 Program blocks from Fig. 2.11 traced through the assembly and loading processes.

2.3.5 Control Sections and Program Linking

- Control sections
  - A part of the program that maintains its *identity* after reassembly
    - Each control section can be loaded and relocated independently
    - Programmer can assemble, load, and manipulate each of these control sections separately
  - Often used for subroutines or other logical subdivisions of a program

2.3.5 Control Sections and Program Linking (Cont.)

- Instruction in one control section may need to refer to instructions or data located in another section
  - Called *external reference*
- However, assembler have no idea where any other control sections will be located at execution time
- The assembler has to generate information for such kind of references, called external references, that will allow the loader to perform the required linking.

Program Blocks v.s. Control Sections

- Program blocks
  - Refer to segments of code that are rearranged with *a single object program unit*
- Control sections
  - Refer to segments that are translated into *independent object program units*

Illustration of Control Sections and Program Linking (Fig 2.15 & 2.16)

First control section: COPY

Implicitly defined as an external symbol

Define external symbols

External reference

Line	Loc	Source statement			Object code
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	

Illustration of Control Sections and Program Linking (Fig 2.15 & 2.16) (Cont.)

Second control section: RDREC

External reference

```
109   0000   RDREC    CSECT
110          .
115          .        SUBROUTINE TO READ RECORD INTO BUFFER
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
```

Illustration of Control Sections and Program Linking (Fig 2.15 & 2.16) (Cont.)

```
0000    WRREC    CSECT
                 .
                 .        SUBROUTINE TO WRITE RECORD FROM BUFFER
                 .
                          EXTREF    LENGTH,BUFFER
0000                      CLEAR     X                  B410
0002                     +LDT       LENGTH             77100000
0006    WLOOP             TD        =X'05'             E32012
0009                      JEQ       WLOOP              332FFA
000C                     +LDCH      BUFFER,X           53900000
0010                      WD        =X'05'             DF2008
0013                      TIXR      T                  B850
0015                      JLT       WLOOP              3B2FEE
0018                      RSUB                         4F0000
                          END       FIRST
001B    *                =X'05'                        05
```

Figure 2.16 Program from Fig. 2.15 with object code.

2.3.5 Control Sections and Program Linking (Cont.)

- **Assembler directive: secname** CSECT
  - Signal the start of a new control section
  - e.g. 109 RDREC CSECT
  - e.g. 193 WRREC CSECT
  - **START** also identifies the beginning of a section
- *External references*
  - References between control sections
  - The assembler generates information for each external reference that will allows the loader to perform the required linking.

External Definition and References

- *External definition*
  - **Assembler directives: EXTDEF name [, name]**
  - EXTDEF names symbols, called ***external symbols***, that are defined in this control section and may be used by other sections
  - Control section names do not need to be named in an EXTDEF statement (e.g., COPY, RDREC, and WRREC)
    - They are automatically considered to be external symbols
- *External reference*
  - **Assembler directives: EXTREF name [,name]**
  - EXTREF names symbols that are used in this control section and are defined elsewhere

2.3.5 Control Sections and Program Linking (Cont.)

- Any instruction whose operand involves an external reference
  - Insert an address of zero and pass information to the loader
    - Cause the proper address to be inserted *at load time*
  - *Relative addressing* is not possible
    - The address of external symbol have no predictable relationship to anything in this control section
    - An *extended format instruction* must be used to provide enough room for the actual address to be inserted

Example of External Definition and References

- Example
  - 15 0003 CLOOP +JSUB RDREC 4B100000
  - 160 0017 +STCH BUFFER,X 57900000
  - 190 0028 MAXLEN WORD BUFEND-BUFFER 000000

How the Assembler Handles Control Sections?

- **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 named by EXTDEF
  - Col. 1 D
  - Col. 2-7 Name of external symbol defined in this section
  - Col. 8-13 Relative address within this control section (hex)
  - Col.14-73 Repeat information in Col. 2-13 for other external symbols
- *Refer record:* lists symbols used as external references, i.e., symbols named by EXTREF
  - Col. 1 R
  - Col. 2-7 Name of external symbol referred to in this section
  - Col. 8-73 Name of other external reference symbols

How the Assembler Handles Control Sections? (Cont.)

- *Modification record* **(revised)**
  - Col. 1 M
  - Col. 2-7 Starting address of the field to be modified (hex)
  - Col. 8-9 Length of the field to be modified, in half-bytes (hex)
  - Col. 10 Modification flag (+ or -)
  - Col.11-16 External symbol whose value is to be added to or subtracted from the indicated field.
- Control section name is automatically an external symbol, it is available for use in Modification records.
- Example (Figure 2.17)
  - M000004^05^+RDREC
  - M000011^05^+WRREC
  - M000024^05^+WRREC
  - M000028^06^+BUFEND //Line 190 BUFEND-BUFFER
  - M000028^06^-BUFFER

Object Program Corresponding to Fig. 2.15 (Fig. 2.17)

```
H^COPY  ^000000^001033
D^BUFFER^000033^BUFEND^001033^LENGTH^00002D
R^RDREC ^WRREC
T^000000^1D^172027^4B100000^032023^290000^332007^4B100000^3F2FEC^032016^0F2016
T^00001D^0D^010003^0F200A^4B100000^3E2000
T^000030^03^454F46
M^000004^05^+RDREC
M^000011^05^+WRREC
M^000024^05^+WRREC
E^000000
```

Object Program Corresponding to Fig. 2.15 (Fig. 2.17) (Cont.)

```
H^RDREC ^000000^00002B
R^BUFFER^LENGTH^BUFEND
T^000000^1D^B410^B400^B440^77201F^E3201B^332FFA^DB2015^A004^332009^57900000^B850
T^00001D^0E^3B2FE9^13100000^4F0000^F1^000000
M^000018^05^+BUFFER
M^000021^05^+LENGTH
M^000028^06^+BUFEND
M^000028^06^-BUFFER
E
```

Object Program Corresponding to Fig. 2.15 (Fig. 2.17) (Cont.)

```
H^WRREC^000000^00001C
R^LENGTH^BUFFER
T^000000^1C^B410^77100000^E32012^332FFA^53900000^DF2008^B850^3B2FEE^4F0000^05
M^000003^05^+LENGTH
M^00000D^05^+BUFFER
E
```

Figure 2.17 Object program corresponding to Fig. 2.15.

Program Linking & Relocation

- Note: the revised Modification record may still be used to perform *program relocation*.
  - E.g. (Fig. 2.8.)
    - M00000705
    - M00001405
    - M00002705 *are changed to*
    -
    - M00000705+COPY //add the beginning address of its section
    - M00001405+COPY
    - M00002705+COPY
- So the same mechanism can be used for program relocation and for program linking.

External References in Expression

- Earlier definitions
  - Required all of the relative terms be paired in an expression (an *absolute expression*), or that all except one be paired (a *relative expression*)
- New restriction
  - Both terms in each pair must be *relative within the same control section*
  - Ex: BUFEND-BUFFER: Legal
  - Ex: RDREC-COPY: illegal

External References in Expression (Cont.)

- However, when an expression involves external references, the assembler cannot determine whether or not the expression is legal.
- How to enforce this restriction
  - The assembler evaluates all of the terms it can, combines these to form an initial expression value, and generates Modification records.
  - The *loader* checks the expression for errors and finishes the evaluation.

2.4 Assembler Design Options

- One-pass assemblers
- Multi-pass assemblers

2.4.1 One-Pass Assemblers

- Goal: avoid a second pass over the source program
- Main problem
  - Forward references to ***data items*** or ***labels on instructions***
- Solution
  - Data items: require all such areas be defined before they are referenced
  - Label on instructions: cannot be eliminated
    - E.g. the logic of the program often requires a forward jump
    - It is too inconvenient if forward jumps are not permitted

Two Types of One-Pass Assemblers:

- Load-and-go assembler
  - Produces object code directly in memory for immediate execution
- The other assembler
  - Produces usual kind of object code for later execution

Load-and-Go Assembler

- No object program is written out, no loader is needed
- Useful for program development and testing
  - Avoids the overhead of writing the object program out and reading it back in
- Both one-pass and two-pass assemblers can be designed as load-and-go
  - However, one-pass also avoids the overhead of an additional pass over the source program
- For a load-and-go assembler, the actual address must be known at assembly time.

Forward Reference Handling in One-pass Assembler

- When the assembler encounter an instruction operand that has not yet been defined:
  1. The assembler omits the translation of operand address
  2. Insert the symbol into SYMTAB, if not yet exist, 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
     1. The forward reference list for that symbol is scanned
     2. The proper address for the symbol is inserted into any instructions previous generated.

Handling Forward Reference in One-pass Assembler (Cont.)

- At the end of the program
  - Any SYMTAB entries that are still marked with * indicate ***undefined symbols***
    - Be flagged by the assembler as errors
  - Search SYMTAB for the symbol named in the END statement and jump to this location to begin execution of the assembled program.

Sample Program for a One-Pass Assembler (Fig. 2.18)

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					

Sample Program for a One-Pass Assembler (Fig. 2.18) (Cont.)

110		.			
115		.	SUBROUTINE TO READ RECORD INTO BUFFE		
120		.			
121	2039	INPUT	BYTE	X'F1'	F1
122	203A	MAXLEN	WORD	4096	001000
124		.			
125	203D	RDREC	LDX	ZERO	041006
130	2040		LDA	ZERO	001006
135	2043	RLOOP	TD	INPUT	E02039
140	2046		JEQ	RLOOP	302043
145	2049		RD	INPUT	D82039
150	204C		COMP	ZERO	281006
155	204F		JEQ	EXIT	30205B
160	2052		STCH	BUFFER,X	54900F
165	2055		TIX	MAXLEN	2C203A
170	2058		JLT	RLOOP	382043
175	205B	EXIT	STX	LENGTH	10100C
180	205E		RSUB		4C0000
195		.			

Sample Program for a One-Pass Assembler (Fig. 2.18) (Cont.)

```
195                    .
200                    .          SUBROUTINE TO WRITE RECORD FROM BUFFER
205                    .
206     2061   OUTPUT     BYTE     X'05'              05
207                    .
210     2062   WRREC      LDX      ZERO               041006
215     2065   WLOOP      TD       OUTPUT             E02061
220     2068              JEQ      WLOOP              302065
225     206B              LDCH     BUFFER,X           50900F
230     206E              WD       OUTPUT             DC2061
235     2071              TIX      LENGTH             2C100C
240     2074              JLT      WLOOP              382065
245     2077              RSUB                        4C0000
255                       END      FIRST
```

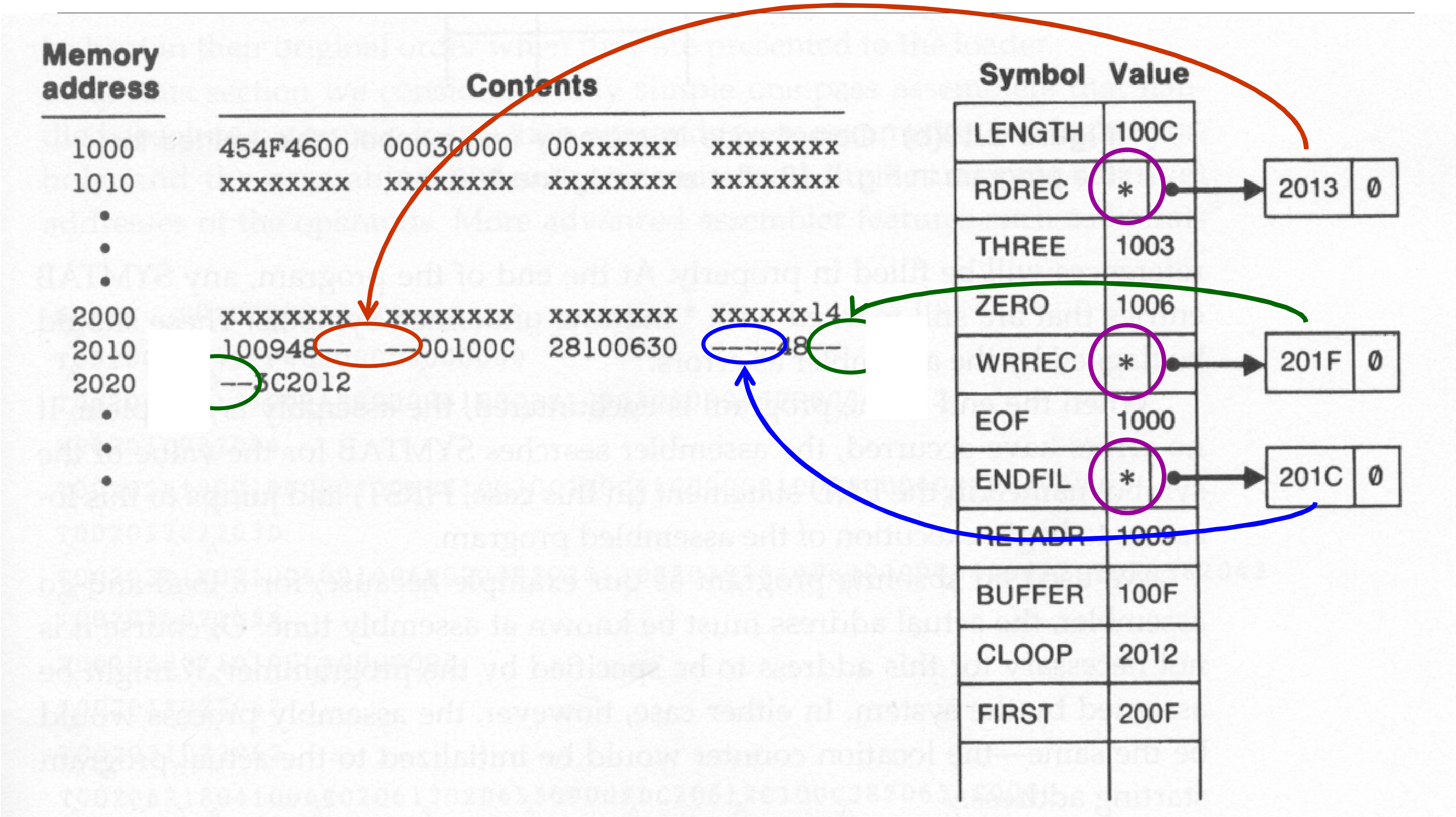
Figure 2.18 Sample program for a one-pass assembler.

Example

- Fig. 2.19 (a)
  - Show the object code in memory and symbol table entries after scanning line 40
  - Line 15: forward reference (RDREC)
    - Object code is marked ----
    - Value in symbol table is marked as * (undefined)
    - Insert *the address of operand* (2013) in a list associated with RDREC
  - Line 30 and Line 35: follow the same procedure

Object Code in Memory and SYMTAB

After scanning line 40

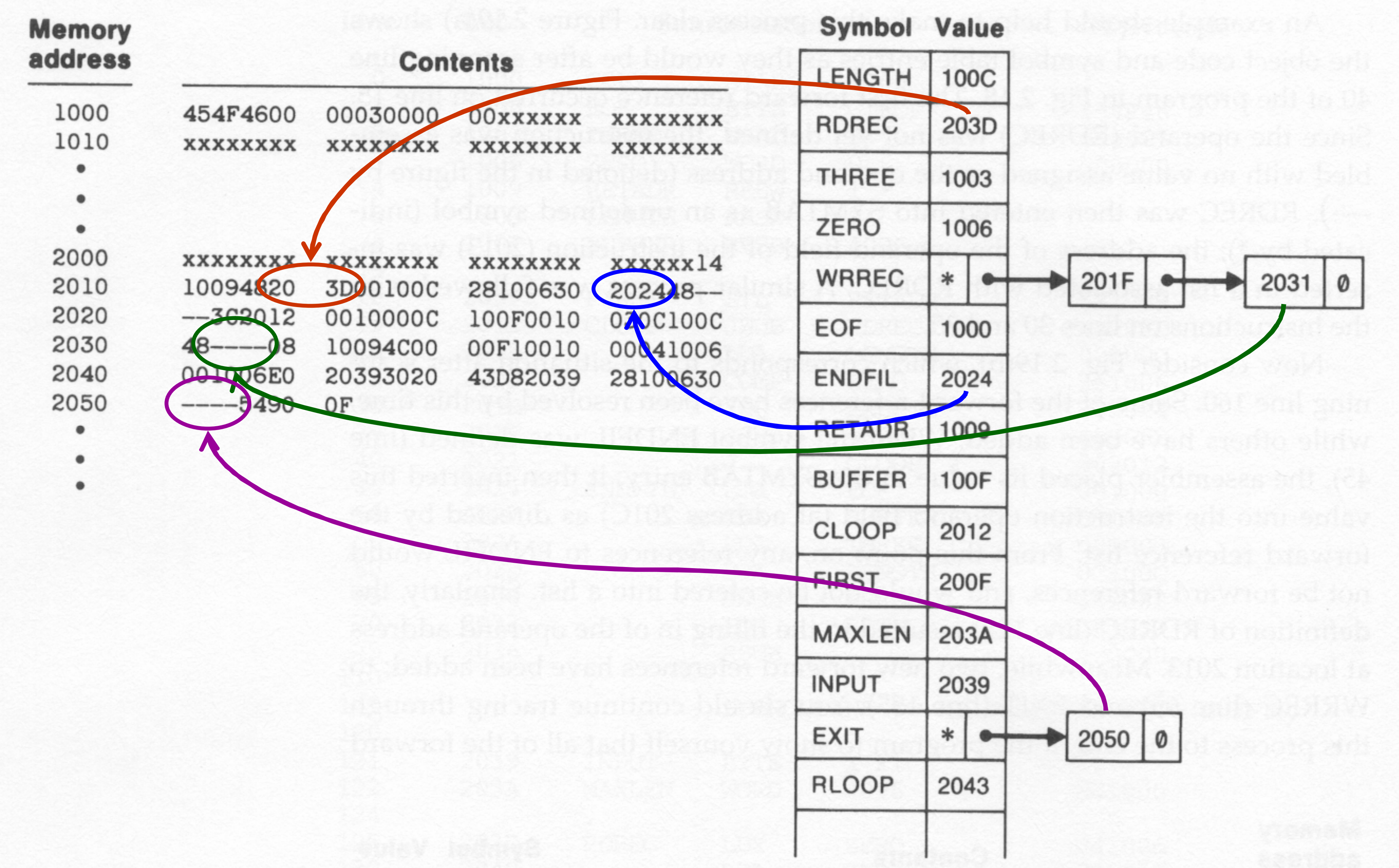


Example (Cont.)

- Fig. 2.19 (b)
  - Show the object code in memory and symbol table entries after scanning line 160
  - Line 45: ENDFIL was defined
    - Assembler place its value in the SYMTAB entry
    - Insert this value into the address (at 201C) as directed by the forward reference list
  - Line 125: RDREC was defined
    - Follow the same procedure
  - Line 65 and 155
    - Two new forward reference (WRREC and EXIT)

Object Code in Memory and SYMTAB

After scanning line 160



Object Code in Memory and SYMTAB Entries for Fig 2.18 (Fig. 2.19b)

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

Symbol	Value
LENGTH	100C
RDREC	203D
THREE	1003
ZERO	1006
WRREC	* → 201F → 2031 Ø
EOF	1000
ENDFIL	2024
RETADR	1009
BUFFER	100F
CLOOP	2012
FIRST	200F
MAXLEN	203A
INPUT	2039
EXIT	* → 2050 Ø
RLOOP	2043

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

One-Pass Assembler Producing Object Code

- Forward reference are entered into the symbol table's list as before
  - If the operand contains an undefined symbol, use 0 as the address and write the Text record to the object program.
- However, when definition of a symbol is encountered, the assembler must generate *another Text record* with the *correct operand address*.
- When the program is loaded, this address will be inserted into the instruction by *loader*.
- The object program records must be kept in their original order when they are presented to the loader

Example

- In Fig. 2.20
  - Second Text record contains the object code generated from lines 10 through 40
    - The operand addressed for the instruction on line 15, 30, 35 have been generated as 0000
  - When the definition of ENDFIL is encountered
    - Generate the third Text record
      - Specify the value 2024 (the address of ENDFIL) is to be loaded at location 201C (the operand field of JEQ in line 30)
      - Thus, the value 2024 will replace the 0000 previously loaded

Object Program from one-pass assembler for Fig 2.18 (Fig 2.20)

```
H^COPY  ^001000^00107A
T^001000^09^454F46^000003^000000
T^00200F^15^141009^480000^00100C^281006^300000^480000^3C2012
T^00201C^02^2024
T^002024^19^001000^0C100F^001003^0C100C^480000^081009^4C0000^F1^001000
T^002013^02^203D
T^00203D^1E^041006^001006^E02039^302043^D82039^281006^300000^54900F^2C203A^382043
T^002050^02^205B
T^00205B^07^10100C^4C0000^05
T^00201F^02^2062
T^002031^02^2062
T^002062^18^041006^E02061^302065^50900F^DC2061^2C100C^382065^4C0000
E^00200F
```

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

2.4.2 Multi-Pass Assemblers

- Motivation: for a 2-pass assembler, any symbol used on the *right-hand side* should be defined previously.
  - No forward references since symbols' value can't be defined during the first pass
    - E.g.

```
APLHA    EQU   BETA
BETA     EQU   DELTA
DELTA    RESW  1
```

Not allowed !

Multi-Pass Assemblers (Cont.)

- Multi-pass assemblers
  - Eliminate the restriction on EQU and ORG
  - Make as many passes as are needed to process the definitions of symbols.
- Implementation
  - To facilitate symbol evaluation, in SYMTAB, each entry must indicates *which symbols are dependent on the values of it*
  - Each entry keeps a <u>*linking list*</u> to keep track of whose symbols' value depend on an this entry

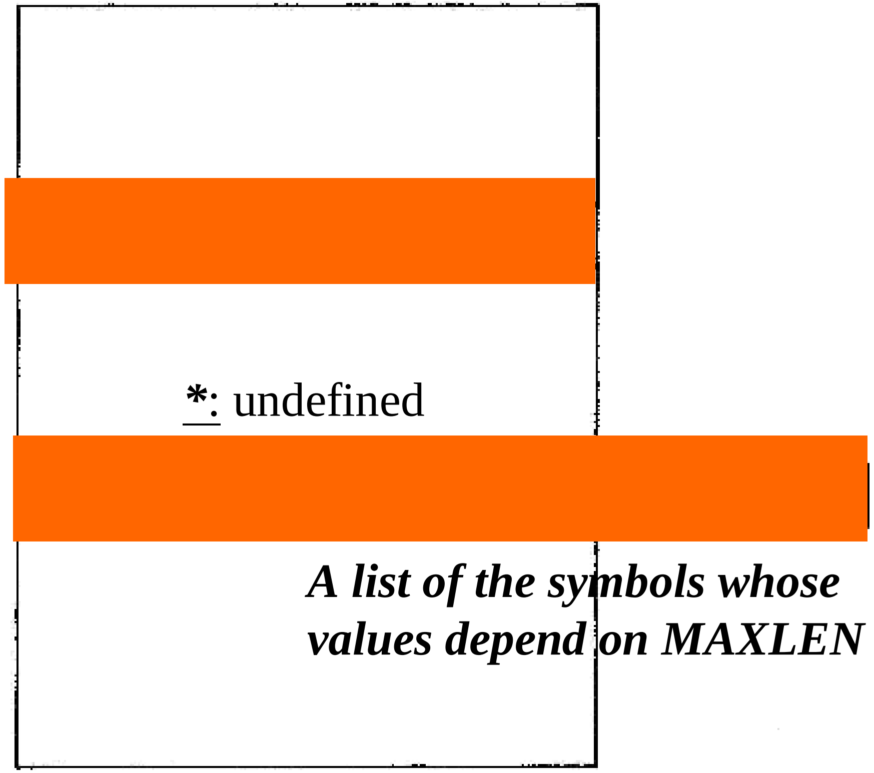
Example of Multi-pass Assembler Operation (fig 2.21a)

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

Example of Multi-Pass Assembler Operation (Fig 2.21b)

&1: one system in the defining expression is undefined

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



*****: undefined

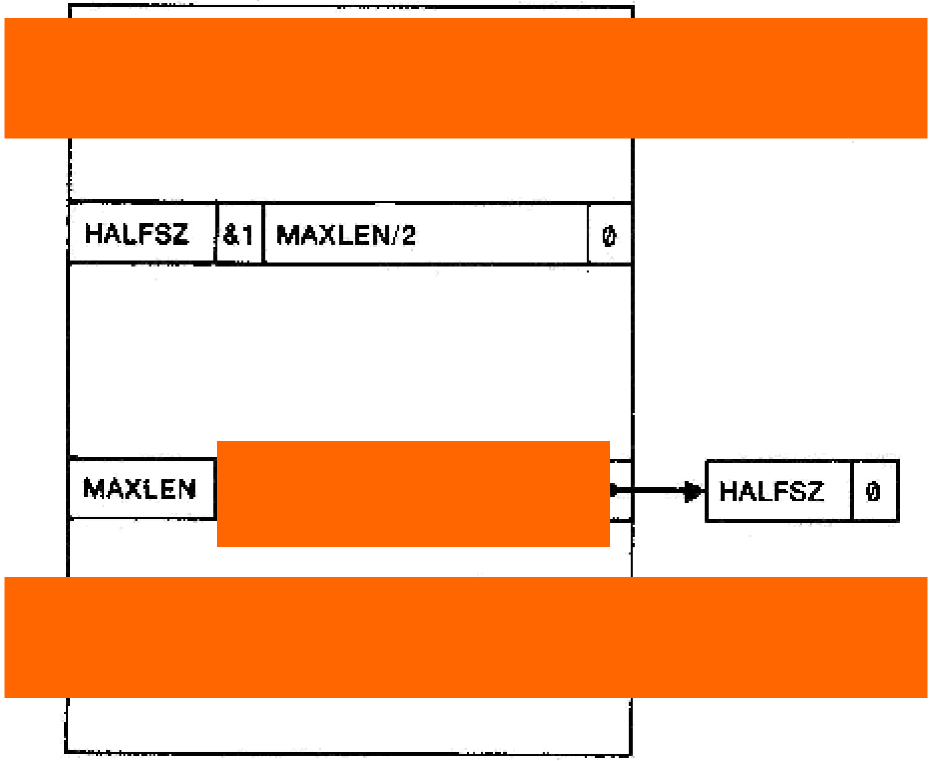
A list of the symbols whose values depend on MAXLEN

(b)

Figure 2.21 Example of multi-pass assembler operation.

Example of Multi-Pass Assembler Operation (Fig 2.21c)

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



(c)

Example of Multi-pass Assembler Operation (fig 2.21d)

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

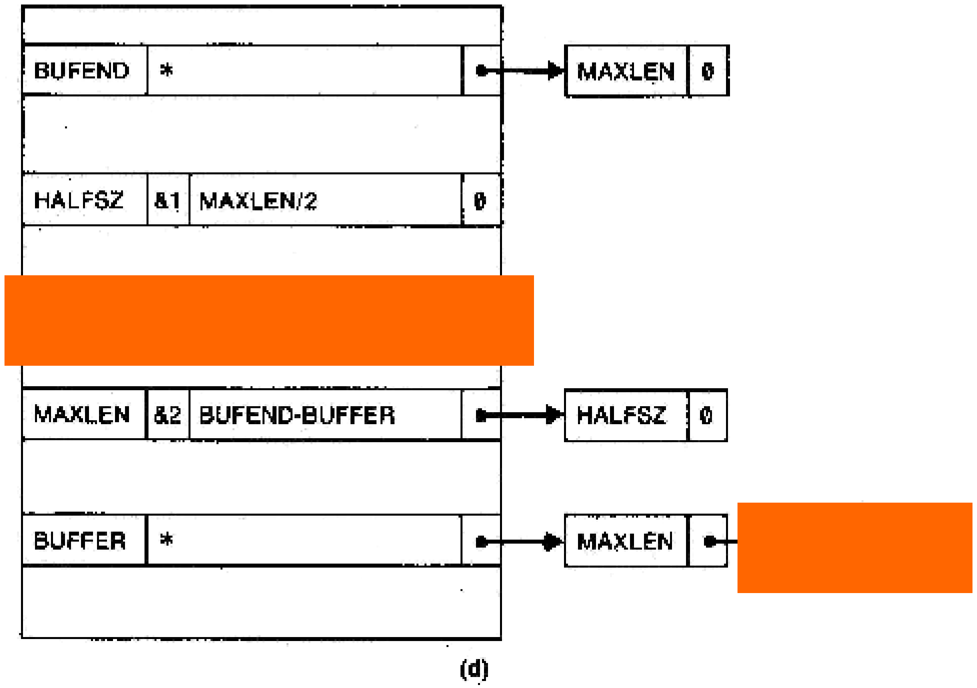
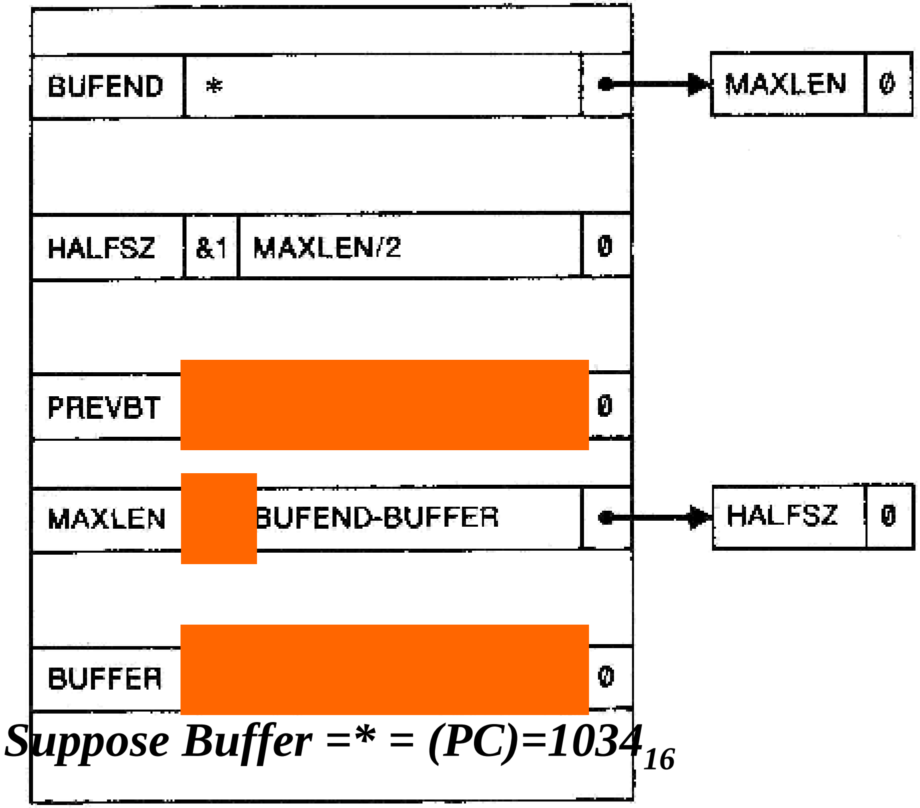


Figure 2.21 (cont'd)

Example of Multi-pass Assembler Operation (fig 2.21e)

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



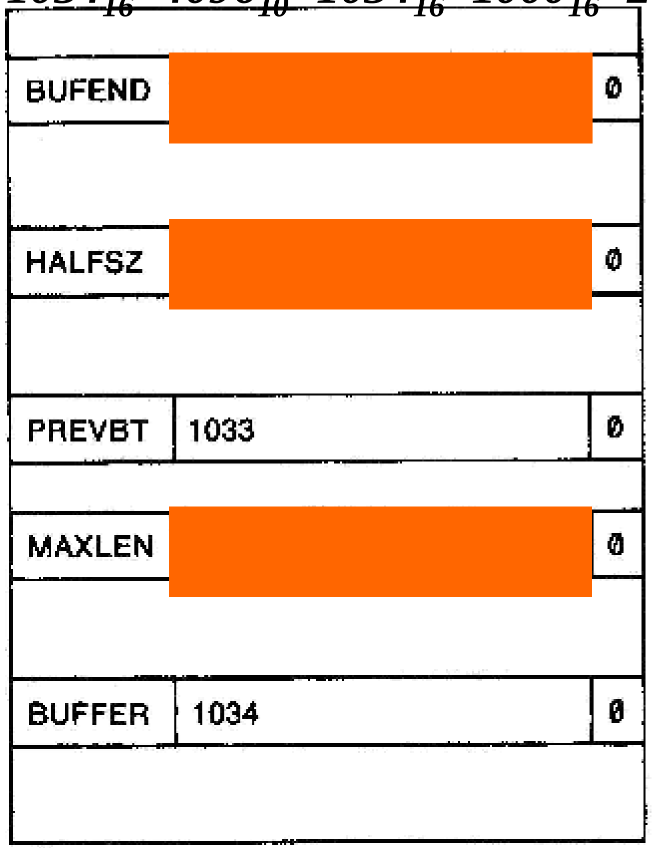
Suppose Buffer =* = (PC)=1034$_{16}$

(e)

Example of Multi-pass Assembler Operation (Fig 2.21f)

$BUFEND=*(PC)=1034_{16}+4096_{10}=1034_{16}+1000_{16}=2034_{16}$

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



(f)

Figure 2.21 *(con'd)*

2.5 Implementation Examples

- Microsoft MASM Assembler
- Sun Sparc Assembler
- IBM AIX Assembler

2.5.1 Microsoft MASM Assembler

- Microsoft MASM assembler for Pentium and other x86 systems
- Programmer of an x86 system views memory as a collection of segments

Microsoft MASM Assembler (Cont.)

- An MASM assembler language program is written as a collection of segments.
- Each segment is defined as belonging to a particular class: CODE, DATA, CONST, STACK
- Assembler directive: **SEGMENT**
  - Similar to program blocks in SIC
  - All parts of a segment are gathered together by assembler
- Segment registers are automatically set by the system loader when a program is loaded for execution: CS (code), SS (stack), DS (data), ES, FS, GS
- Assembler directive: **ASSUME**
  - By default, assembler assumes all references to data segments use register DS
  - We can change by the assembler directive ASSUME
  - e.g. ASSUME ES:DATASEG2
    - Tell the assembler that register ES indicate the segment DATASEG2
    - Thus, any reference to labels are defined in DATASEG2 will be assembled using register ES
  - Similar to **BASE** directive in SIC/XE
    - BASE tell a SIC/XE assembler the contents of register B
    - ASSUME tell MASM the contents of a segment register

Microsoft MASM Assembler (Cont.)

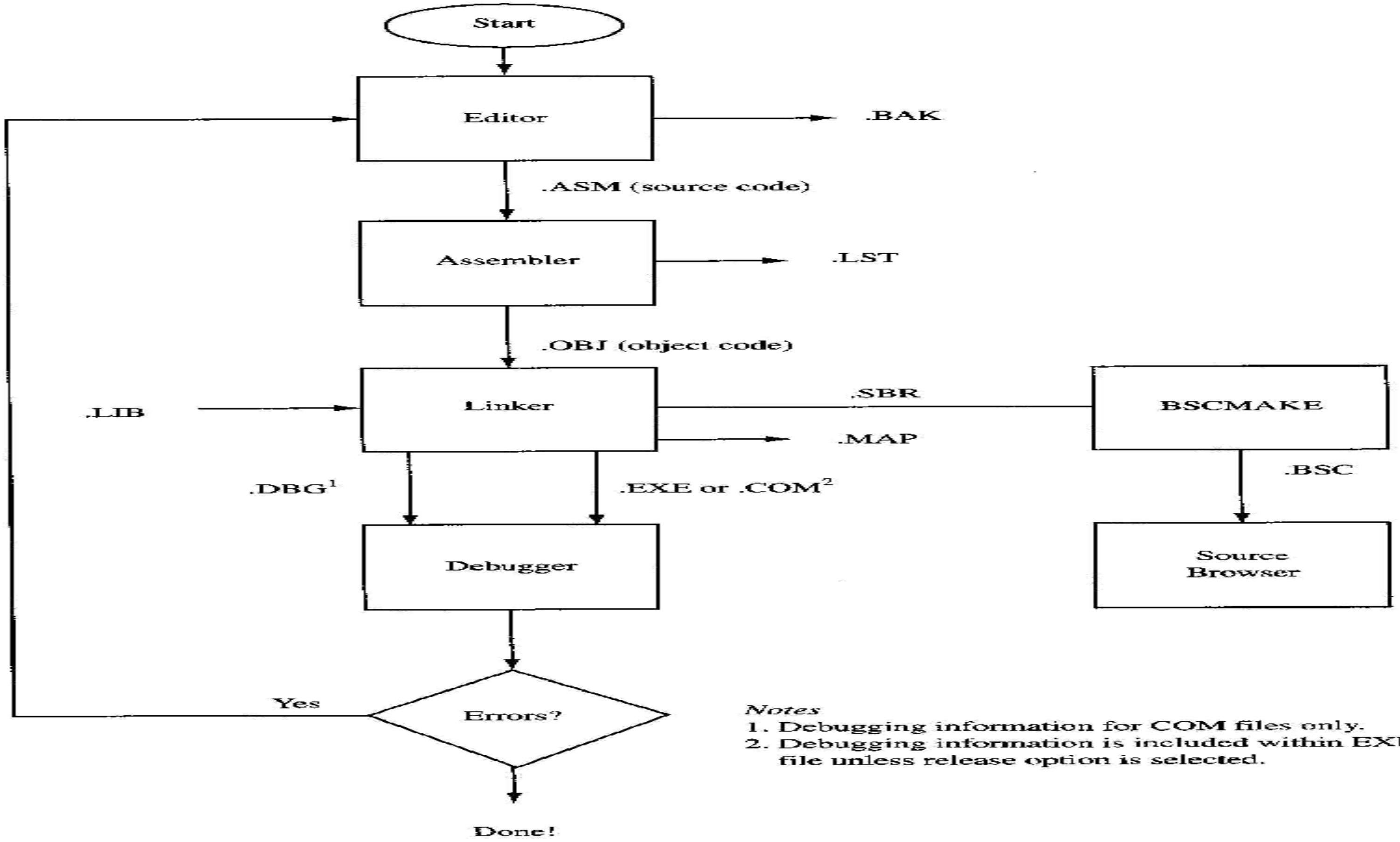
- Jump instructions are assembled in 2 different ways:
  - Near jump: jump to a target in the same code segment
    - 2- or 3-byte instruction
  - Far jump: jump to a target in a different code segment
    - 5-byte instruction
- Problem: Jump with forward reference
  - By default, MASM assumes that a forward jump is a near jump
  - If it is a far jump, ***programmer*** must tell the assembler
    - E.g. JMP ***FAR PTR*** TARGET
- In x86, the ***length*** of an assembled instruction depends on *the operands that are used*.
  - Operands maybe registers, memory locations, immediate values (1~4 bytes)
  - Thus, Pass 1 in MASM is much complex that in SIC assembler

Microsoft MASM Assembler (Cont.)

- External references between separately assembled modules must be handled by the *linker*
  - MASM directive: **PUBLIC**, **EXTRN**
  - Similar to **EXTDEF**, **EXTREF** in SIC/XE
- The object program from MASM may be in several different formats to allow easy and efficient execution of the program in a variety of operating environments.

Development of 80x86 Assembly Language Program

Figure 6.1 The development of an assembly language program requires four steps: edit, assemble, link, and debug. Several different file types are created in the process. The three-letter extension associated with each is shown.



Notes
1. Debugging information for COM files only.
2. Debugging information is included within EXE file unless release option is selected.

Template for an 80x86 Assembly Program

Figure 6.14 Template for an 80x86 assembly language program in EXE form.

```
        Page lines,columns
        Title

       .CPU

;*******************************************
;*              Program Name              *
;*                                        *
;*   Put a description of the program here.*
;*******************************************


; Program equates
var1            equ     xx
var2            equ     xx

;*******************************************
;*              Stack Segment              *
;*                                         *
;*******************************************
sseg            segment         stack
                db              100 dup (?)                     ;100 byte stack
sseg            ends


;*******************************************
;*              Data  Segment              *
;*                                         *
;*******************************************
dseg            segment

first           db              ?                               ;One byte
second          db              100     dup   (?)               ;100 bytes
ASCII           db              'Put ASCII messages inside quotes'

dseg            ends
```

(a)

Figure 6.14 *(continued)*

```
;************************************
;*            Code Segment          *
;*                                  *
;************************************

cseg         segment          'code'
assume       cs:cseg, ss:sseg, ds:dseg
             mov              ax,dseg                ;Load DS
             mov              ds,ax
start:                                               ;Main program begins here


;Place procedures in the code segment

procedure1   proc
             pusha
                                                     ;Body of procedure

             popa
procedure1   endp


procedure2   proc
             pusha
                                                     ;Body of procedure

             popa
procedure2   endp


cseg         ends                                    ;End of code segment
             end              start                  ;End of program
```

(b)

A Simple 80x86 Assembly Program

Figure 6.2 Program 5.1 rewritten in assembly language form.

```
          Page      58,132
          Title     Program 5.1

;*********************************************
;            Display ASCII Character Set      *
;*                                            *
;*   This program displays all of the ASCII  *
;*   characters with codes 0-127.             *
;*                                            *
;*********************************************

cseg      segment   'code'
          assume    cs:cseg, ds:cseg, ss:cseg, es:cseg

          org       100h              ;Leave room for PSP

start:    mov       ax,0002           ;BIOS service 0, video mode 2
          int       10h               ;Set video mode and clear screen
          mov       ah,2              ;BIOS service 2
          mov       dx,0a00h          ;Row 10, column 0
          mov       bh,0              ;Page 0
          int       10h               ;Position cursor
          mov       ax,0e00h          ;BIOS service 0E, first character is 0
IIA:      int       10h               ;Print character
          inc       al                ;Next
          cmp       al,80h            ;Done?
          jnz       IIA               ;No: loop again
          int       20h               ;Yes: back to DOS

cseg      ends
          end       start
```

The list File

Figure 6.3 The list file shows (a) the object code associated with each instruction and (b) a summary list of the segments and symbols used in the program.

```
Microsoft (R) Macro Assembler Version 6.11
Program 5.1                                                        Page 1 - 1


                                                   Page     58,132
                                                   Title    Program 5.1

                                            ;*******************************************
                                            ;          Display ASCII Character Set     *
                                            ;*                                         *
                                            ;*   This program displays all of the ASCII *
                                            ;*   characters with codes 0-127.          *
                                            ;*                                         *
                                            ;*******************************************

0000                                        cseg     segment 'code'
                                                     assume  cs:cseg, ds:cseg, ss:cseg, es:cseg

                                                     org     100h             ;Leave room for PSP

0100   B8 0002                    start:    mov      ax,0002          ;BIOS service 0, video mode 2
0103   CD 10                                int      10h              ;Set video mode and clear screen
0105   B4 02                                mov      ah,2             ;BIOS service 2
0107   BA 0A00                              mov      dx,0a00h         ;Row 10, column 0
010A   B7 00                                mov      bh,0             ;Page 0
010C   CD 10                                int      10h              ;Position cursor
010E   B8 0E00                              mov      ax,0e00h         ;BIOS service 0E, first character is 0
0111   CD 10                      IIA:      int      10h              ;Print character
0113   FE C0                                inc      al               ;Next
0115   3C 80                                cmp      al,80h           ;Done?
0117   75 F8                                jnz      IIA              ;No: loop again
0119   CD 20                                int      20h              ;Yes: back to DOS

011B                                        cseg     ends
                                                     end     start
```

(a)

The list File (Cont.)

Figure 6.3(b) *(continued)*

```
Segments and Groups:

                N a m e              Size    Length   Align   Combine Class

cseg . . . . . . . . . . . . . .   .16 Bit   011B    Para      Private 'CODE'



Symbols:

                N a m e              Type    Value    Attr

IIA  . . . . . . . . . . . . . .   .L Near   0111    cseg
start  . . . . . . . . . . . .   . .L Near   0100    cseg

          0 Warnings
          0 Errors
```

(b)

An Example of 80x86 Assembly Program

Figure 6.7 Program 5.3 written in EXE form: (a) the stack and data segments; (b) the code segment; (c) The display procedure; and (d) The program summary.

```
                                        Page43,132
                                        Title    Program 5.3

                                        .8086

                                  ;*******************************************************
                                  ;*              8-Bit BCD Adder                        *
                                  ;*                                                     *
                                  ;* This program inputs two packed BCD numbers from     *
                                  ;* the keyboard, computes their sum and outputs the    *
                                  ;* result to the screen.                               *
                                  ;*                                                     *
                                  ;* Example: The user types: 62+34-                     *
                                  ;* The computer responds:   96                         *
                                  ;*******************************************************

                                  ;************************
                                  ;*     Stack Segment    *
                                  ;************************
0000                              sseg     segment  stack
0000   0020 [                              db       32 dup   (?)        ;32 bytes for stack
        00
       ]
0020                              sseg     ends

                                  ;************************
                                  ;*     Data Segment     *
                                  ;************************
0000                              dseg     segment
0000   0008 [                     buff     db       8 dup    (?)        ;8 byte input buffer
00
]
0008                              dseg     ends
```

(a)

Figure 6.7 *(continued)*

```
                                    ;************************
                                    ;      Code Segment     *
                                    ;************************
0000                                        cseg    segment  'code'
                                            assume  cs:cseg, ds:dseg, ss:sseg

0000   B8 ---- R            main:   mov     ax,dseg                 ;Get address of data segment
0003   8E D8                        mov     ds,ax                   ;and store in DS

0005   8D 16 0000 R                 lea     dx,buff                 ;Point DX at input buffer
0009   B4 0A                        mov     ah,0ah                  ;DOS function 0AH
000B   8B F2                        mov     si,dx                   ;Point SI at input buffer
000D   C6 04 08                     mov     byte ptr [si],8         ;8 byte buffer
0010   CD 21                        int     21h                     ;Get the two numbers
0012   B4 0E                        mov     ah,0eh                  ;BIOS video service
0014   B0 0A                        mov     al,0ah                  ;ASCII line feed
0016   CD 10                        int     10h
0018   80 6C 02 30                  sub     byte ptr [si+2],30h     ;Convert each digit to BCD
001C   80 6C 03 30                  sub     byte ptr [si+3],30h
0020   80 6C 05 30                  sub     byte ptr [si+5],30h
0024   80 6C 06 30                  sub     byte ptr [si+6],30h
0028   B1 04                        mov     cl,4                    ;Four rotates
002A   D2 44 03                     rol     byte ptr [si+3],cl      ;Form LSD
002D   D2 44 06                     rol     byte ptr [si+6],cl
0030   D3 4C 02                     ror     word ptr [si+2],cl      ;Add to MSD
0033   D3 4C 05                     ror     word ptr [si+5],cl
0036   8A 44 03                     mov     al,[si+3]               ;Fetch first BCD number
0039   02 44 06                     add     al,[si+6]               ;Add to second
003C   27                           daa                             ;Keep results decimal
003D   8A F8                        mov     bh,al                   ;Save results
003F   73 05                        jnc     IIIB                    ;Check for hundredths digit
0041   B0 01                        mov     al,1                    ;Set hundredths digit
0043   E8 0009                      call    dspy                    ;Display it
0046   8A C7                IIIB:   mov     al,bh                   ;Recover low order result
0048   E8 0002                      call    dspy                    ;Display low order result
004B   B4 4C                        mov     ah,4ch                  ;Terminate
004D   CD 21                        int     21h                     ;Return to DOS
```

(b)

(continued on next page)

Figure 6.7 *(continued)*

```
                          ;***********************
                          ;*  Display Procedure    *
                          ;***********************
                          ;    Function:   Display Two Digit BCD Number
                          ;    Inputs:     BCD number in AL
                          ;    Outputs:    None
                          ;    Calls:      BIOS interrupt 10H
                          ;    Destroys:   AX, BL, CL, flags

004F                      dspy    proc                        ;Display procedure
004F   8A D8                      mov       bl,al             ;Save original number
0051   24 F0                      and       al,0f0h           ;Force bits 0-3 low
0053   B1 04                      mov       cl,4              ;Four rotates
0055   D2 C8                      ror       al,cl             ;Rotate MSD into LSD
0057   04 30                      add       al,30h            ;Convert to ASCII
0059   B4 0E                      mov       ah,0eh            ;BIOS video service 0E
005B   CD 10                      int       10h               ;Display character
005D   8A C3                      mov       al,bl             ;Recover original number
005F   24 0F                      and       al,0fh            ;Force bits 4-7 low
0061   04 30                      add       al,30h            ;Convert to ASCII
0063   CD 10                      int       10h               ;Display character
0065   C3                         ret                         ;Return to calling program
0066                      dspy    endp


0064                      cseg    ends
                                  end       main
```

(c)

Figure 6.7 *(continued)*

```
Segments and Groups:

                N a m e            Size     Length   Align   Combine Class

cseg . . . . . . . . . . . . . .  16 Bit   0066     Para    Private 'CODE'
dseg . . . . . . . . . . . . . .  16 Bit   0008     Para    Private
sseg . . . . . . . . . . . . . .  16 Bit   0020     Para    Stack



Procedures,  parameters and locals:

                N a m e            Type     Value    Attr

dspy . . . . . . . . . . . . . .  P Near   004F     cseg   Length= 0017 Public


Symbols:

                N a m e            Type     Value    Attr

IIIB . . . . . . . . . . .   . .L Near   0046     cseg
buff . . . . . . . . . . .   . .Byte     0000     dseg
main . . . . . . . . . . . . . .  L Near   0000     cseg

           0 Warnings
           0 Errors
```

(d)

Memory Map of the Example Program

Figure 6.9 Memory map for PROG53.EXE. The specific segment addresses are found by loading the program with Debug and then viewing the CPU registers with the R command.