Unit 2: Assembler Design [20 hrs]

2.1: BASIC ASSEMBLER FUNCTIONS

- Some assembler directives are:
- **START**: Specify name and starting address for the program.
- **END**: Indicate the end of the source program and (optionally) specify the first executable instruction in the program.
- **BYTE**: Generate character or hexadecimal constant, occupying as many bytes as needed to represent the constant.
- **WORD**: Generate one-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.

- The line numbers are for reference only and are not part of the program. These numbers also help to relate corresponding parts of different versions of the program.
- Indexed addressing is indicated by adding the modifier *“X" following the operand.*
- *Lines beginning* with "." contain comments only.
- The program contains a main routine that reads records from an input device (identified with device code F1) and copies them to an output device (code 05).
- This main routine calls subroutine RDREC to read a record into a buffer and subroutine WRREC to write the record from the buffer to the out-put device.
- Each subroutine must transfer the record one character at a time because the only I/O instructions available are RD and WO.
- The buffer is necessary because the I/O rates for the two devices, such as a disk and a slow printing terminal, may be very different.
- 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; thus, the RSUB will return control to the operating system.

An assembler language program for the basic version of SIC (2.1)

Line **Source statement**

```
COPY      START   1000            COPY FILE FROM INPUT TO OUTPUT
FIRST     STL     RETADR          SAVE RETURN ADDRESS
CLOOP     JSUB    RDREC           READ INPUT RECORD
          LDA     LENGTH          TEST FOR EOF (LENGTH = 0)
          COMP    ZERO
          JEQ     ENDFIL          EXIT IF EOF FOUND
          JSUB    WRREC           WRITE OUTPUT RECORD
          J       CLOOP           LOOP
ENDFIL    LDA     EOF             INSERT END OF FILE MARKER
          STA     BUFFER
          LDA     THREE           SET LENGTH = 3
          STA     LENGTH
          JSUB    WRREC           WRITE EOF
          LDL     RETADR          GET RETURN ADDRESS
          RSUB                    RETURN TO CALLER
EOF       BYTE    C'EOF'
THREE     WORD    3
ZERO      WORD    0
RETADR    RESW    1
LENGTH    RESW    1               LENGTH OF RECORD
BUFFER    RESB    4096            4096-BYTE BUFFER AREA
.
```

(2.1 Continued)

```
        .         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
        .
        .         SUBROUTINE TO WRITE RECORD FROM BUFFER
        .
        WRREC     LDX       ZERO              CLEAR LOOP COUNTER
        WLOOP     TD        OUTPUT            TEST OUTPUT DEVICE
                  JEQ       WLOOP             LOOP UNTIL READY
                  LDCH      BUFFER,X          GET CHARACTER FROM BUFFER
                  WD        OUTPUT            WRITE CHARACTER
                  TIX       LENGTH            LOOP UNTIL ALL CHARACTERS
                  JLT       WLOOP               HAVE BEEN WRITTEN
                  RSUB                        RETURN TO CALLER
        OUTPUT    BYTE      X'05'             CODE FOR OUTPUT DEVICE
                  END       FIRST
```

A Simple 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.

Program for 2.1 with Object Code (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	
110		.			

(2.2 Continued)

```
                .       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
                .
                .       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
```

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.
- Because of this, most assemblers make two passes over the source program.
- The first pass does little more than scan the source program for label definitions and assign addresses.

- In addition to translating the instructions of the source program, the assembler must process statements called ***assembler directives*** *(or* ***pseudo-instructions****).*
- These statements are not translated into machine instructions (although they may have an effect on the object program).
- Instead, they provide instructions to the assembler itself.
- Examples of assembler directives are statements like BYTE and WORD, which direct the assembler to generate constants as part of the object program, and 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 the program.

- Finally, the assembler must write the generated object code onto some out- put 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**.
- 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.

Header record:

Col. 1	H
Col. 2-7	Program name
Col. 8-13	Starting address of object program (hexadecimal)
Col. 14-19	Length of object program in bytes (hexadecimal)

Text record:

Col. 1	T
Col. 2-7	Starting address for object code in this record (hexadecirnal)
Col. 8-9	Length of object code in this record in bytes (hexadecimal)
Col. 10—69	Object code, represented in hexadecimal (2 columns per byte of object code)

End record:

Col. 1	E
Col. 2-7	Address of first executable instruction in object program (hexadecimal)

The object program for 2.2 (2.3)

```
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
```

Pass 1 (define symbols):

1. Assign addresses to all statements in the program.
2. Save the values (addresses) assigned to all labels for use in Pass 2.
3. Perform some processing of assembler directives. (This includes processing that affects address assignment, such as determining the length of data areas defined by BYTE, RESW, etc.)

Pass 2 (assemble instructions and generate object program):

1. Assemble instructions (translating operation codes and looking up addresses).
2. Generate data values defined by BYTE, WORD, etc.
3. Perform processing of assembler directives not done during Pass 1.
4. Write the object program and the assembly listing.

Assembler Algorithm and Data Structures

- Our simple assembler uses two major internal data structures: the **Operation Code Table** (**OPTAB**) and the **Symbol Table** (**SYMTAB**).
- **OPTAB** is used to look up mnemonic operation codes and translate them to their machine language equivalents.
- **SYMTAB** is used to store values (addresses) assigned to labels.
- We also need a **Location Counter LOCCTR**. This 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**.
- Thus whenever we reach a label in the source program, the current value of **LOCCTR** gives the address to be associated with that label.

- The Operation Code Table (**OPTAB**) must contain (at least) the mnemonic operation code and its machine language equivalent.
- In more complex assemblers, this table also contains information about instruction format and length.
- During Pass 1, **OPTAB** is used to look up and validate operation codes in the source program.
- In Pass 2, it is used to translate the operation codes to machine language.
- Actually, in our simple SIC assembler, both of these processes could be done together in either Pass 1 or Pass 2.
- During Pass 1 of the assembler, labels are entered into **SYMTAB** as they are encountered in the source program, along with their assigned 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.

Algorithm for Pass 1 Assembler (2.4 a)

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
```

(2.4 a Continued)

```
        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}
```

Algorithm for Pass 2 Assembler (2.4 b)

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
```

(2.4 b Continued)

```
                                    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)
```

2.2: MACHINE-DEPENDENT ASSEMBLER FEATURES

- In our assembler language, indirect addressing is indicated by adding the prefix @ to the operand(see line 70).
- Immediate operands are denoted with the prefix # (lines 25, 55, 133).
- 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.
- 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.
- The extended instruction format is specified with the prefix + added to the operation code in the source statement (see lines 15, 35, 65).
- It is the programmer's responsibility to specify this form of addressing when it is required.

Example of SIC/XE Program for 2.1 (2.5)

```
Line       Source statement

  5     COPY      START   0           COPY FILE FROM INPUT TO OUTPUT
 10     FIRST     STL     RETADR      SAVE RETURN ADDRESS
 12               LDB     #LENGTH     ESTABLISH BASE REGISTER
 13               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     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
 80     EOF       BYTE    C'EOF'
 95     RETADR    RESW    1
100     LENGTH    RESW    1           LENGTH OF RECORD
105     BUFFER    RESB    4096        4096-BYTE BUFFER AREA
110     .
```

(2.5 Continued)

```
        .
        .           SUBROUTINE TO READ RECORD INTO BUFFER
        .
        RDREC       CLEAR    X                CLEAR LOOP COUNTER
                    CLEAR    A                CLEAR A TO ZERO
                    CLEAR    S                CLEAR S TO ZERO
                   +LDT     #4096
        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
        INPUT       BYTE     X'F1'            CODE FOR INPUT DEVICE
        .
        .           SUBROUTINE TO WRITE RECORD FROM BUFFER
        .
        WRREC       CLEAR    X                CLEAR LOOP COUNTER
                    LDT      LENGTH
        WLOOP       TD       OUTPUT           TEST OUTPUT DEVICE
                    JEQ      WLOOP            LOOP UNTIL READY
                    LDCH     BUFFER,X         GET CHARACTER FROM BUFFER
                    WD       OUTPUT           WRITE CHARACTER
                    TIXR     T                LOOP UNTIL ALL CHARACTERS
                    JLT      WLOOP              HAVE BEEN WRITTEN
                    RSUB                      RETURN TO CALLER
        OUTPUT      BYTE     X'05'            CODE FOR OUTPUT DEVICE
                    END      FIRST
```

Instruction Formats and Addressing Modes [Program form 2.5 with Object Code (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	
110		.			

(2.6 Continued)

```
          .          SUBROUTINE TO READ RECORD INTO BUFFER
          .
1036      RDREC      CLEAR     X                   B410
1038                 CLEAR     A                   B400
103A                 CLEAR     S                   B440
103C                +LDT      #4096                75101000
1040      RLOOP      TD        INPUT               E32019
1043                 JEQ       RLOOP               332FFA
1046                 RD        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
          .
          .          SUBROUTINE TO WRITE RECORD FROM 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
```

- Translation of register-to-register instructions such as **CLEAR** (line 125) and **COMPR** (line 150) presents no new problems.
- The assembler must simply convert the mnemonic operation code to machine language (using **OPTAB**) and change each register mnemonic to its numeric equivalent.
- This translation is done during Pass 2, at the same point at which the other types of instructions are assembled.
- The conversion of register mnemonics to numbers can be done with a separate table; however, it is often convenient to use the symbol table for this purpose.
- To do this, **SYMTAB** would be preloaded with the register names (A, X, etc.) and their values (0, 1, etc.).
- Most of the register-to-memory instructions are assembled using either program-counter relative or base relative addressing.
- 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).
- Of course, the 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).

- 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.
- This 4-byte format contains a 20-bit address field, which is large enough to contain the full memory address.
- In this case, there is no displacement to be calculated.
- For example, in the instruction shown below, the operand address is 1036. This full address is stored in the instruction, with bit *e set to 1 to indicate extended instruction format.*

15 0006 CLOOP +JSUB RDREC 4B101036

- The instruction

  10 0000 FIRST STL RETADR 17202D

  is a typical example of program-counter relative assembly.
- During execution of instructions on SIC (as in most computers), the program counter is advanced *after each instruction is fetched and before it is executed.*
- *Thus 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 would, of course, 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 byte 17 instead of 14.

- Another example of program-counter relative assembly is the instruction

  40 0017 J CLOOP 3F2FEC

- Here the operand address is 0006.
- During instruction execution, the program counter will contain the address 0001A.
- Thus the displacement required is 6 -1A = -14.
- This is represented (using 2's complement for negative numbers) in a 12-bit field as FEC, which is the displacement assembled into the object code.

- The instruction

  160 104E STCH BUFFER, X 57C003

  is a typical example of base relative assembly.
- 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 must be 36 - 33 = 3.
- Notice that bits *x and b are set to 1 in the assembled instruction* to indicate indexed and base relative addressing.
- Another example is the instruction **STX LENGTH** on line **175**. Here the displacement calculated is 0.

- The instruction

  55 0020 LDA #3 010003

  is a typical example of this, with the operand stored in the instruction as 003, and bit *i set to 1 to indicate immediate addressing.*

- *Another example can be* found in the instruction

  133 103C +LDT #4096 75101000

- In this case the 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.)

- A different way of using immediate addressing is shown in the instruction

  12 0003 LDB #LENGTH 69202D

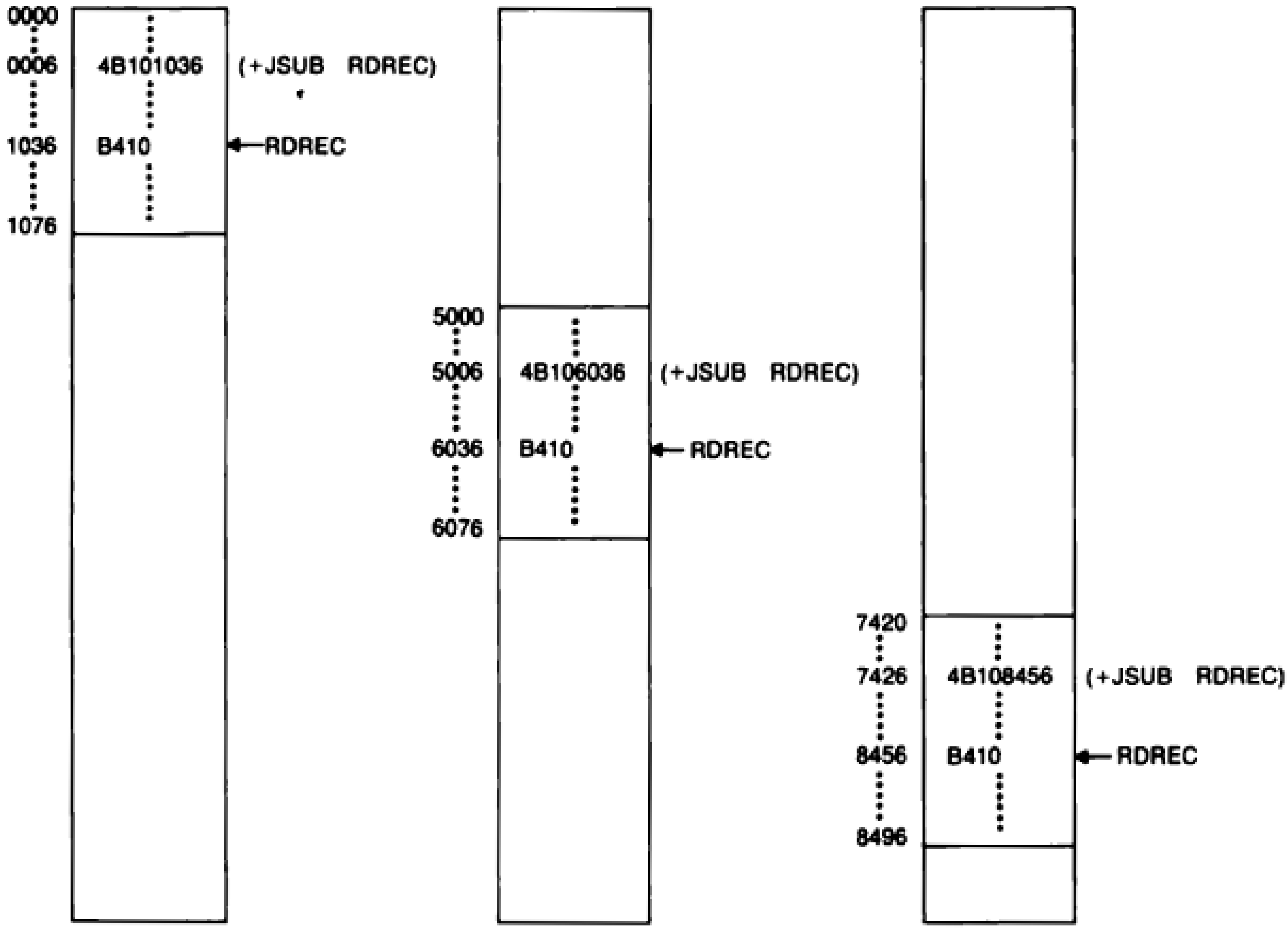
- In this statement the immediate operand is the *symbol* ***LENGTH.***
- *Since the value of this symbol is the address assigned to it, this immediate instruction has* the effect of loading register B with the address of **LENGTH**.
- Note here that we have combined program-counter relative addressing with immediate addressing.
- Although this may appear unusual, the interpretation is consistent with our previous uses of immediate operands.
- In general, the target address calculation is performed; then, if immediate mode is specified, the *target address* (not the *contents stored at that address) becomes the operand.*
- *(In the* LDA statement on line 55, for example, bits *x, b, and p are all 0. Thus the target* address is simply the displacement 003.)

Program Relocation

- It is often desirable to have more than one program at a time sharing the memory and other resources of the machine.
- If we knew in advance exactly which programs were to be executed concurrently in this way, we could assign addresses when the programs were assembled so that they would fit together without overlap or wasted space.
- Most of the time, however, it is not practical to plan program execution this closely. (We usually do not know exactly when jobs will be submitted, exactly how long they will run, etc.)
- Because of this, it is desirable to be able to load a program into memory wherever there is room for it.
- In such a situation the actual starting address of the program is not known until load time.

- Since the assembler does not know the actual location where the program will be loaded, it cannot make the necessary changes in the addresses used by the program.
- However, the assembler can identify for the loader those parts of the object program that need modification.
- An object program that contains the information necessary to perform this kind of modification is called a ***relocatable program***.
- To look at this in more detail, consider the program from 2.5 and 2.6. In the preceding section, we assembled this program using a starting address of 0000.

Examples of Program Relocation (2.7)



(a) (b) (c)

- Figure 2.7(a) shows this program loaded beginning at address 0000.
- The **JSUB** instruction from line 15 is loaded at address 0006.
- The address field of this instruction contains 01036, which is the address of the instruction labelled **RDREC**. (These addresses are, of course, the same as those assigned by the assembler.)
- Now suppose that we want to load this program beginning at address 5000, as shown in Fig. 2.7(b).
- The address of the instruction labelled **ROREC** is then 6036.
- Thus the **JSUB** instruction must be modified as shown to contain this new address.
- Likewise, if we loaded the program beginning at address 7420 (Fig. 2.7c), the **JSUB** instruction would need to be changed to 48108456 to correspond to the new address of **RDREC**.

- Note that no matter where the program is 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:

1. When the assembler generates the object code for the JSUB instruction we are considering, it will insert the address of **RDREC** *relative to the start of the program. (This is the reason we initialized the location* counter to 0 for the assembly.)
2. The assembler will also produce a command for the loader, instructing it to *add the beginning address of the program to the address* field in the **JSUB** instruction at load time.

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

Modification record:

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

- The length is stored in half-bytes (rather than bytes) because the address field to be modified may not occupy an integral number of bytes.
- (For example, the address field in the **JSUB** instruction we considered above occupies 20 bits, which is 5 half-bytes.)

- By now it should be clear that the only parts of the program that require modification at load time are those that specify direct (as opposed to relative) addresses.
- For this SIC/XE program, the only such direct addresses are found in extended format (4-byte) instructions.
- This is an advantage of relative addressing - if we were to attempt to relocate the program from Fig. 2.1, we would find that almost every instruction required modification.

Object Program for 2.6 (2.7)

```
HCOPY  ^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
```

2.3: MACHINE-INDEPENDENT ASSEMBLER FEATURES

- In this section, we discuss some common assembler features that are not closely related to machine architecture.
- Of course, more advanced machines tend to have more complex software; therefore the features we consider are more likely to be found on larger and more complex machines.
- However, the presence or absence of such capabilities is much more closely related to issues such as programmer convenience and software environment than it is to machine architecture.

Literals

- It is often convenient for the programmer 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.*
- The use of literals is illustrated by the program in Fig. 2.9.
- The object code generated for the statements of this program is shown in Fig. 2.10. (This program is a modification of the one in Fig. 2.5).

Program demonstrating additional assembler features (2.9)

Line	Source statement			
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
110	.			

(2.9 Continued)

```
.               SUBROUTINE TO READ RECORD INTO BUFFER
.
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
INPUT   BYTE    X'F1'           CODE FOR INPUT DEVICE
.
.               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
```

Program from Fig. 2.9 with object code (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		.			

(2.10 C0ntinued)

```
                .          SUBROUTINE TO READ RECORD INTO BUFFER
                .
1036    RDREC      CLEAR     X                   B410
1038               CLEAR     A                   B400
103A               CLEAR     S                   B440
103C              +LDT      #MAXLEN              75101000
1040    RLOOP      TD        INPUT               E32019
1043               JEQ       RLOOP               332FFA
1046               RD        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
                .
                .          SUBROUTINE TO WRITE RECORD FROM BUFFER
                .
105D    WRREC      CLEAR     X                   B410
105F               LDT       LENGTH              774000
1062    WLOOP      TD       =X'05'               E32011
1065               JEQ       WLOOP               332FFA
1068               LDCH      BUFFER,X            53C003
106B               WD       =X'05'               DF2008
106E               TIXR      T                   B850
1070               JLT       WLOOP               3B2FEF
1073               RSUB                          4F0000
                   END       FIRST
1076    *         =X'05'                         05
```

- In our assembler language notation, a literal is identified with the prefix *=*, which is followed by a specification of the literal value, using the same notation as in the BYTE statement.
- Thus the literal in the statement

  45 001A ENDFIL LDA =C'EOF‘ 032010

  specifies a 3-byte operand whose value is the character string EOF.
- Likewise the statement

  215 1062 WLOOP TD =X'05’ E32011

  specifies a 1-byte literal with the hexadecimal value 05.
- The notation used for literals varies from assembler to assembler; however, most assemblers use some symbol (as we have used **=**) to make literal identification easier.

- It is important to understand the difference between a literal and an immediate operand.
- With immediate addressing, the operand value is assembled as part of the machine instruction.
- With a literal, the assembler generates the specified value as a constant at some other memory location.
- The *address of* this generated constant is used as the target address for the machine instruction.
- The effect of using a literal is exactly the same as if the programmer had defined the constant explicitly and used the label assigned to the constant as the instruction operand.
- (In fact, the generated object code for lines 45 and 215 in Fig. 2.10 is identical to the object code for the corresponding lines in Fig. 2.6.)
- Compare the object instructions generated for lines 45 and 55 in Fig. 2.10 to make sure you understand how literals and immediate operands are handled.

- All of the literal operands used in a program are gathered together into one or more ***literal pools****.*
- *Normally literals are placed into a pool at the end of* the program.
- The assembly listing of a program containing literals usually includes a listing of this literal pool, which shows the assigned addresses and the generated data values.
- Such a literal pool listing is shown in Fig. 2.10 immediately following the **END** statement.
- In this case, the pool consists of the single literal **=X'05'**.
- In some cases, however, it is desirable to place literals into a pool at some other location in the object program.
- To allow this, we introduce the assembler directive **LTORG** (line 93 in Fig. 2.9).
- When the assembler encounters a **LTORG** statement, it creates a literal pool that contains all of the literal operands used since the previous **LTORG** (or the beginning of the program).
- This literal pool is placed in the object program at the location where the **LTORG** directive was encountered (see Fig. 2.10).
- Of course, literals placed in a pool by LTORG will not be repeated in the pool at the end of the program.

- The basic data structure needed is a *literal table* ***LITTAB.***
- *For each literal used,* this table contains the literal name, the operand value and length, and the address assigned to the operand when it is placed in a literal pool.
- **LITTAB** is often organized as a hash table, using the literal name or value as the key.
- As each literal operand is recognized during Pass 1, the assembler searches **LITTAB** for the specified literal name (or value).
- If the literal is already present in the table, no action is needed; if it is not present, the literal is added to **LITTAB** (leaving the address unassigned).
- When Pass 1 encounters a LTORG statement or the end of the program, the assembler makes a scan of the literal table.
- At this time each literal currently in the table is assigned an address (unless such an address has already been filled in).
- As these addresses are assigned, the location counter is updated to reflect the number of bytes occupied by each literal.
- During Pass 2, the-operand address for use in generating object code is obtained by searching **LITTAB** for each literal operand encountered.
- 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.
- If a literal value represents an address in the program (for example, a location counter value), the assembler must also generate the appropriate Modification record.

Symbol-Defining Statements

- Most assemblers provide an assembler directive that allows the programmer to define symbols and specify their values.
- The assembler directive generally used is **EQU** (for "equate").
- The general form of such a statement is

  symbol EQU value
- This statement defines the given symbol (i.e., enters it into **SYMTAB**) and assigns to it the value specified.
- The value may be given as a constant or as any expression involving constants and previously defined symbols.
- One common use of **EQU** is to establish symbolic names that can be used for improved readability in place of numeric values.
- For example, on line 133 of the program in Fig. 2.5 we used the statement

  +LDT #4096

  to load the value 4096 into register T.
- This value represents the maximum length record we could read with subroutine **RDREC**. The meaning is not, however, as clear as it might be.
- If we include the statement

  MAXLEN EQU 4096

  in the program, we can write line 133 as

  +LDT #MAXLEN

- Another common use of **EQU** is in defining mnemonic names for registers.
- We have assumed that our assembler recognizes standard mnemonics for registers -A, X, L, etc.
- Suppose, however, that the assembler expected register *numbers instead of names in an instruction like RMO. This would require the* programmer to write (for example) RMO 0,1 instead of RMO A,X.
- In such a case the programmer could include a sequence of EQU statements like

A	**EQU**	**0**
X	**EQU**	**1**
L	**EQU**	**2**

- These statements cause the symbols A, X, L,... to be entered into **SYMTAB** with their corresponding values 0, 1,2, ...
- An instruction like **RMO A,X** would then be allowed.
- The assembler would search **SYMTAB**, finding the values 0 and 1 for the symbols A and X, and assemble the instruction.

- Consider, however, a machine that has general-purpose registers.
- These registers are typically designated by 0, 1, 2,... (or RO, R1, R2,...).
- In a particular program, however, some of these may be used as base registers, some as index registers, some as accumulators, etc.
- Furthermore, this usage of registers changes from one program to the next.
- By writing statements like

  BASE EQU R1

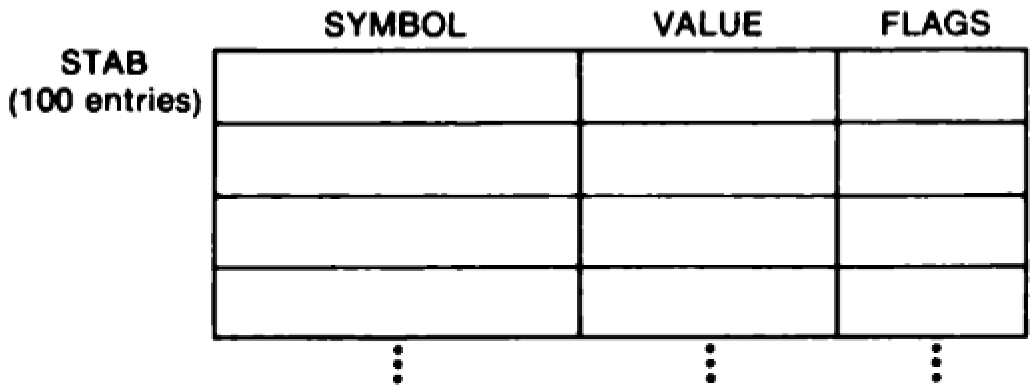
  COUNT EQU R2

  INDEX EQU R3

  the programmer can establish and use names that reflect the logical function of the registers in the program.

- There is another common assembler directive that can be used to indirectly assign values to symbols.
- This directive is usually called **ORG** (for "origin").
- Its form is: **ORG value**

  where *value is a constant or an expression involving constants and previously* defined symbols.
- When this statement is encountered during assembly of a program, the assembler resets its location counter (**LOCCTR**) to the specified value.
- Since the values of symbols used as labels are taken from **LOCCTR**, the **ORG** statement will affect the values of all labels defined until the next **ORG**.



- In this table, the **SYMBOL** field contains a 6-byte user-defined symbol; **VALUE** is a one-word representation of the value assigned to the symbol; **FLAGS** is a 2-byte field that specifies symbol type and other information.
- We could reserve space for this table with the statement

STAB RESB 1100

- We want to be able to refer to the fields **SYMBOL**, **VALUE**, and **FLAGS** individually, so we must also define these labels.
- One way of doing this would be with **EQU** statements:

SYMBOL EQU STAB

VALUE EQU STAB+6

FLAGS EQU STAB+9

- We can accomplish the same symbol definition using ORG in the following way:

STAB	**RESB**	**1100**
	ORG	**STAB**
SYMBOL	**RESB**	**6**
VALUE	**RESW**	**1**
FLAGS	**RESB**	**2**
	ORG	**STAB+11 00**

- The first **ORG** resets the location counter to the value of **STAB** (i.e., the beginning address of the table).
- The label on the following **RESB** statement defines **SYMBOL** to have the current value in **LOCCTR**; this is the same address assigned to **SYMTAB**.
- **LOCCTR** is then advanced so the label on the **RESW** statement assigns to **VALUE** the address (STAB+6), and so on.
- The result is a set of labels with the same values as those defined with the **EQU** statements above.
- This method of definition makes it clear, however, that each entry in **STAB** consists of a 6-byte **SYMBOL**, followed by a one-word **VALUE**, followed by a 2-byte FLAGS.
- The last **ORG** statement is very important.
- It sets **LOCCTR** back to its previous value-the address of the next unassigned byte of memory after the table **STAB**.
- This is necessary so that any labels on subsequent statements, which do not represent part of **STAB**, are assigned the proper addresses.

Expressions

- Assemblers generally allow arithmetic expressions formed according to the normal rules using the operators +, -, *, and /.
- Division is usually defined to produce an integer result.
- Individual terms in the expression may be constants, user-defined symbols, or special terms.
- The most common such special term is the current value of the location counter (often designated by *****).
- This term represents the value of the next unassigned memory location.
- Thus in Fig. 2.9 the statement

  106 BUFEND EQU *

  gives **BUFEND** a value that is the address of the next byte after the buffer area.

- Expressions are classified as either ***absolute expressions*** *or* ***relative expressions*** depending upon the type of value they produce.
- An expression that contains only absolute terms is, of course, an **absolute expression**.
- However, absolute expressions may also contain relative terms provided the relative terms occur in pairs and the terms in each such pair have opposite signs.
- It is not necessary that the paired terms be adjacent to each other in the expression; however, all relative terms must be capable of being paired in this way.
- None of the relative terms may enter into a multiplication or division operation.
- A **relative expression** is one in which all of the relative terms except one can be paired as described above; the remaining unpaired relative term must have a positive sign.
- As before, no relative term may enter into a multiplication or division operation.
- Expressions that do not meet the conditions given for either absolute or relative expressions should be flagged by the assembler as errors.

- Consider, for example, the program of Fig. 2.9. In the statement

  107 MAXLEN EQU BUFEND-BUFFER

  both **BUFEND** and **BUFFER** are relative terms, each representing an address within the program.
- However, the expression represents an absolute value: the *difference between the two addresses, which is the length of the buffer area in* bytes.
- Notice that the assembler listing in Fig. 2.10 shows the value calculated for this expression (hexadecimal 1000) in the Loc column.
- This value does not represent an address, as do most of the other entries in that column.
- However, it does show the value that is associated with the symbol that appears in then source statement (**MAXLEN**).

Program Blocks

- In all of the examples so far the program being assembled was treated as a unit.
- The source programs logically contained subroutines, data areas, etc.
- However, they were handled by the assembler as one entity, resulting in a single block of object code.
- Within this object program the generated machine instructions and data appeared in the same order as they were written in the source program.
- Many assemblers provide features that allow more flexible handling of the source and object programs.
- Some features allow the generated machine instructions and data to appear in the object program in a different order from the corresponding source statements.
- Other features result in the creation of several independent parts of the object program.
- These parts maintain their identity and are handled separately by the loader.
- We use the term ***program blocks*** *to refer to segments of code that are rearranged within a single object* program unit, and ***control sections*** *to refer to segments that are translated into* independent object program units.
- In this section we consider the use of program blocks and how they are handled by the assembler.

- Figure 2.11 shows our example program as it might be written using program blocks. In this case three blocks are used.
- The first (**unnamed**) program block contains the executable instructions of the program.
- The second (named **CDATA**) contains all data areas that are a few words or less in length.
- The third (named **CBLKS**) contains all data areas that consist of larger blocks of memory.
- The assembler directive **USE** indicates which portions of the source program belong to the various blocks.
- At the beginning of the program, 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.
- The **USE** statement on line 92 signals the beginning of the block named **CDATA**.
- Source statements are associated with this block until the **USE** statement on line 103, which begins the block named **CBLKS**.
- The **USE** statement may also indicate a continuation of a previously begun block.
- Thus the statement on line 123 resumes the default block, and the statement on line 183 resumes the block named **CDATA**.

Example of a program with multiple program blocks (2.11)

Line	Source statement			
5	COPY	START	0	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	#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
92		USE	CDATA	
95	RETADR	RESW	1	
100	LENGTH	RESW	1	LENGTH OF RECORD
103		USE	CBLKS	
105	BUFFER	RESB	4096	4096-BYTE BUFFER AREA
106	BUFEND	EQU	*	FIRST LOCATION AFTER BUFFER
107	MAXLEN	EQU	BUFEND-BUFFER	MAXIMUM RECORD LENGTH
110	.			

(2.11 Continued)

```
        .          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
        .
        .          SUBROUTINE TO WRITE RECORD FROM BUFFER
        .
                   USE
        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
                   USE       CDATA
                   LTORG
                   END       FIRST
```

- The assembler accomplishes this logical rearrangement of code by maintaining, during Pass 1, a separate location counter for each program block.
- The location counter for a block is initialized to 0 when the block is first begun.
- The current value of this location counter is saved when switching to another block, and the saved value is restored when resuming a previous block.
- Thus during Pass 1 each label in the program is assigned an address that is relative to the start of the block that contains it.
- When labels are entered into the symbol table, the block name or number is stored along with the assigned relative address.
- 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 (beginning with relative location 0) (Fig 2.12 b*****).
- For code generation during Pass 2, the assembler needs the address for each symbol relative to the start of the object program (not the start of an individual program block).
- This is easily found from the information in **SYMTAB**.
- The assembler simply adds the location of the symbol, relative to the start of its block, to the assigned block starting address (2.12 c*****).
- **[* - See additional files]**

Program from Fig. 2.11 with object code (2.12 a)

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	
110			.			

(2.12 a Continued)

```
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                     .
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
```

See additional doc for 2.12 b and 2.12 c

- Figure 2.12 a demonstrates this process applied to our sample program.
- The column headed **loc/Block** shows the relative address (within a program block) assigned to each source line and a block number indicating which program block is involved (0 = **default block**, 1 = **CDATA**, 2 = **CBLKS**).
- This is essentially the same information that is stored in **SYMTAB** for each symbol.
- Notice that the value of the symbol **MAXLEN** (line 107) is shown without a block number. This indicates that **MAXLEN** is an absolute symbol, whose value is not relative to the start of any program block.
- At the end of Pass 1 the assembler constructs a table that contains the starting addresses and lengths for all blocks.
- For our sample program, this table looks like

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

- 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 can simply write the object code as it is generated during Pass 2 and insert the proper load address in each Text record.
- These load addresses will, of course, reflect the starting address of the block as well as the relative location of the code within the block.
- This process is illustrated in Fig. 2.13.
- The first two Text records are generated from the source program lines 5 through 70.
- When the **USE** statement on line 92 is recognized, the assembler writes out the current Text record (even though there is still room left in it).
- The assembler then prepares to begin a new Text record for the new program block.
- As it happens, the statements on lines 95 through 105 result in no generated code, so no new Text records are created.

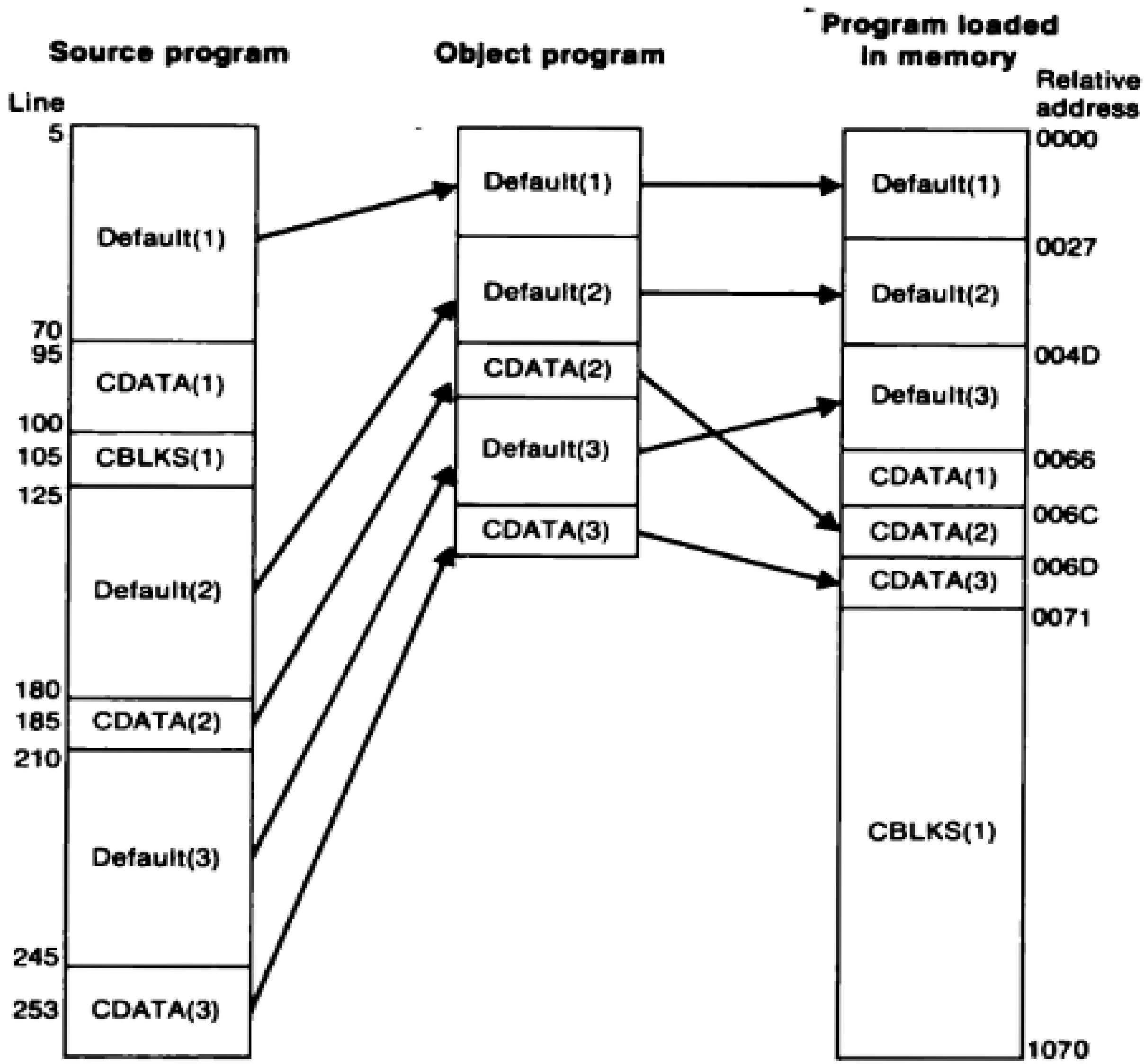
- The next two Text records come from lines 125 through 180.
- This time the statements that belong to the next program block do result in the generation of object code.
- The fifth Text record contains the single byte of data from line 185.
- The sixth Text record resumes the default program block and the rest of the object program continues in similar fashion.

```
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.

- The loader will simply load the object code from each record at the indicated address.
- When this loading is completed, the generated code from the default block will occupy relative locations 0000 through 0065; the generated code and reserved storage for **CDATA** will occupy locations 0066 through 0070; and the storage reserved for **CBLKS** will occupy locations 0071 through 1070.
- Figure 2.14 traces the blocks of the example program through this process of assembly and loading.
- Notice that the program segments marked **CDATA(1)** and **CBLKS(1)** are not actually present in the object program.
- Because of the way the addresses are assigned, storage will automatically be reserved for these areas when the program is loaded.

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



Control Sections and Program Linking

- A ***control section*** *is a part of the program that maintains its* identity after assembly; each such control section can be loaded and relocated independently of the others.
- Different control sections are most often used for subroutines or other logical subdivisions of a program.
- The programmer can assemble, load, and manipulate each of these control sections separately.
- The resulting flexibility is a major benefit of using control sections.

- When control sections form logically related parts of a program, it is necessary to provide some means for ***linking them together***.
- *For example, instructions* in one control section might need to refer to instructions or data located in another section.
- Because control sections are independently loaded and relocated, the assembler is unable to process these references in the usual way.
- The assembler has no idea where any other control section will be located at execution time.
- Such references between control sections are called ***external references.***
- The assembler generates information for each external reference that will allow the loader to perform the required linking.
- In this section we describe how external references are handled by our assembler.
- Chapter 3 discusses in detail how the actual linking is performed.

- Figure 2.15 shows our example program as it might be written using multiple control sections.
- In this case there are three control sections: one for the main program and one for each subroutine.
- The **START** statement identifies the beginning of the assembly and gives a name (**COPY**) to the first control section.
- The first section continues until the **CSECT** statement on line 109.
- This assembler directive signals the start of a new control section named **RDREC**.
- Similarly, the **CSECT** statement on line 193 begins the control section named **WRREC**.
- The assembler establishes a separate location counter (beginning at 0) for each control section, just as it does for program blocks.

- Control sections differ from program blocks in that they are handled separately by the assembler. (It is not even necessary for all control sections in a program to be assembled at the same time.)
- Symbols that are defined in one control section may not be used directly by another control section; they must be identified as external references for the loader to handle.
- Figure 2.15 shows the use of two assembler directives to identify such references: **EXTDEF** (**external definition**) and **EXTREF** (**external reference**).
- The **EXTDEF** statement in a control section names symbols, called ***external symbols***, *that are defined in this* control section and may be used by other sections.
- Control section names (in this case **COPY**, **RDREC**, and **WRREC**) do not need to be named in an **EXTDEF** statement because they are automatically considered to be external symbols.
- The **EXTREF** statement names symbols that are used in this control section and are defined elsewhere.
- For example, the symbols **BUFFER**, **BUFEND**, and **LENGTH** are defined in the control section named **COPY** and made available to the other sections by the **EXTDEF** statement on line 6.
- The third control section (**WRREC**) uses two of these symbols, as specified in its **EXTREF** statement (line 207).
- The order in which symbols are listed in the **EXTDEF** and **EXTREF** statements is not significant.

Illustration of control sections and program linking (2.15)

```
Line        Source statement

  5     COPY      START    0                  COPY FILE FROM INPUT TO OUTPUT
  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
```

(2.15 Continued)

```
RDREC   CSECT
.
.       SUBROUTINE TO READ RECORD INTO BUFFER
.
        EXTREF  BUFFER,LENGTH,BUFEND
        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
INPUT   BYTE    X'F1'           CODE FOR INPUT DEVICE
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 BUFFER
        WD     =X'05'           WRITE CHARACTER
        TIXR    T               LOOP UNTIL ALL CHARACTERS
        JLT     WLOOP             HAVE BEEN WRITTEN
        RSUB                    RETURN TO CALLER
        END     FIRST
```

Program from Fig. 2.15 with object code (2.16)

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	
109	0000	RDREC	CSECT		
110		.			

(2.16 Continued)

```
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

193       0000        WRREC     CSECT
195                   .
200                   .         SUBROUTINE TO WRITE RECORD FROM BUFFER
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
```

- Figure 2.16 shows the generated object code for each statement in the program.
- Consider first the instruction

  15 0003 CLOOP +JSUB RDREC 4B100000
- The operand (**RDREC**) is named in the **EXTREF** statement for the control section, so this is an external reference.
- The assembler has no idea where the control section containing **RDREC** will be loaded, so it cannot assemble the address for this instruction.
- Instead the assembler inserts an address of zero and passes information to the loader, which will cause the proper address to be inserted at load time.
- The address of **RDREC** will have no predictable relationship to anything in this control section; therefore relative addressing is not possible.
- Thus an extended format instruction must be used to provide room for the actual address to be inserted.
- This is true of any instruction whose operand involves an external reference.

- Similarly, the instruction

  160 0017 +STCH BUFFER, X 57900000

  makes an external reference to **BUFFER**.
- The instruction is assembled using extended format with an address of zero.
- The *x bit is set to 1 to indicate indexed* addressing, as specified by the instruction.
- The statement

  190 0028 MAXLEN WORD BUFEND- BUFFER 000000

  is only slightly different.
- Here the value of the data word to be generated is specified by an expression involving two external references: **BUFEND** and **BUFFER**.
- As before, the assembler stores this value as zero. When the program is loaded, the loader will add to this data area the address of **BUFEND** and subtract from it the address of **BUFFER**, which results in the desired value.
- Note the difference between the handling of the expression on line 190 and the similar expression on line 107.
- The symbols **BUFEND** and **BUFFER** are defined in the same control section with the **EQU** statement on line 107.
- Thus the value of the expression can be calculated immediately by the assembler.
- This could not be done for line 190; **BUFEND** and **BUFFER** are defined in another control section, so their values are unknown at assembly time.

- The two new record types an: **Define** and **Refer**.
- A **Define record** gives information about external symbols that are defined in this control section-that is, symbols named by **EXTDEF**.
- A **Refer record** lists symbols that are used as external references by the control section-that is, symbols named by **EXTREF**.
- The formats of these records are as follows.

Define record:

Col. 1	D
Col. 2-7	Name of external symbol defined in this control section
Col. 8-13	Relative address of symbol within this control section (hexadecimal)
Col. 14-73	Repeat information in Col. 2-13 for other external symbols

Refer record:

Col. 1	R
Col. 2-7	Name of external symbol referred to in this control section
Col. 8-73	Names of other external reference symbols

- The other information needed for program linking is added to the **Modification record** type.
- The new format is as follows.

Modification record (revised):

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

- The first three items in this record are the same as previously discussed.
- The two new items specify the modification to be performed: adding or subtracting the value of some external symbol.
- The symbol used for modification may be defined either in this control section or in another one.

Object program corresponding to Fig. 2.15 (2.17)

```
HCOPY  ^000000^001033
DBUFFER^000033^BUFEND^001033^LENGTH^00002D
RRDREC ^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
```

```
HRDREC ^000000^00002B
RBUFFER^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
```

```
HWRREC ^000000^00001C
RLENGTH^BUFFER
T^000000^1C^B410^77100000^E32012^332FFA^53900000^DF2008^B850^3B2FEE^4F000005
M^000003^05^+LENGTH
M^00000D^05^+BUFFER
E
```

2.4: ASSEMBLER DESIGN OPTIONS

- In this section we discuss two alternatives to the standard two-pass assembler logic.
- One-pass assemblers are used when it is necessary or desirable to avoid a second pass over the source program.
- Multipass assembler, an extension to the two-pass logic that allows an assembler to handle forward references during symbol definition.

One-Pass Assemblers

- It is easy to eliminate forward references to data items; we can simply require that all such areas be defined in the source program before they are referenced.
- This restriction is not too severe.
- The programmer merely places all storage reservation statements at the start of the program rather than at the end.
- Unfortunately, forward references to labels on instructions cannot be eliminated as easily. The logic of the program often requires a forward jump for example, in escaping from a loop after testing some condition.
- Requiring that the programmer eliminate all such forward jumps would be much too restrictive and inconvenient.
- Therefore, the assembler must make some special provision for handling forward references.
- To reduce the size of the problem, many one-pass assemblers do, however, prohibit (or at least discourage) forward references to data items.

- There are two main types of one-pass assembler.
- One type produces object code directly in memory for immediate execution; the other type produces the usual kind of object program for later execution.
- We use the program in Fig. 2.18 to illustrate our discussion of both types.
- This example is the same as in Fig. 2.2, with all data item definitions placed ahead of the code that references them.
- The generated object code shown in Fig. 2.18 is for reference only; we will discuss how each type of one-pass assembler would actually generate the object program required.

Sample program for a one-pass assembler (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

(2.18 Continued)

```
                .       SUBROUTINE TO READ RECORD INTO BUFFER
                .
        2039    INPUT   BYTE    X'F1'           F1
        203A    MAXLEN  WORD    4096            001000
                .
        203D    RDREC   LDX     ZERO            041006
        2040            LDA     ZERO            001006
        2043    RLOOP   TD      INPUT           E02039
        2046            JEQ     RLOOP           302043
        2049            RD      INPUT           D82039
        204C            COMP    ZERO            281006
        204F            JEQ     EXIT            30205B
        2052            STCH    BUFFER,X        54900F
        2055            TIX     MAXLEN          2C203A
        2058            JLT     RLOOP           382043
        205B    EXIT    STX     LENGTH          10100C
        205E            RSUB                    4C0000
                .
                .       SUBROUTINE TO WRITE RECORD FROM BUFFER
                .
        2061    OUTPUT  BYTE    X'05'           05
                .
        2062    WRREC   LDX     ZERO            041006
        2065    WLOOP   TD      OUTPUT          E02061
        2068            JEQ     WLOOP           302065
        206B            LDCH    BUFFER,X        50900F
        206E            WD      OUTPUT          DC2061
        2071            TIX     LENGTH          2C100C
        2074            JLT     WLOOP           382065
        2077            RSUB                    4C0000
                        END     FIRST
```

- We first discuss one-pass assemblers that generate their object code in memory for immediate execution. No object program is written out, and no loader is needed.
- This kind of ***load-and-go*** *assembler is useful in a system that* is oriented toward program development and testing.
- A **load-and-go** assembler avoids the overhead of writing the object program out and reading it back in.
- This can be accomplished with either a one- or a two-pass assembler. However, a one-pass assembler also avoids the overhead of an additional pass over the source program.
- Because the object program is produced in memory rather than being written out on secondary storage, the handling of forward references becomes less difficult.
- The assembler simply generates object code instructions as it scans the source program.
- If an instruction operand is a symbol that has not yet been defined, the operand address is omitted when the instruction is assembled.
- The symbol used as an operand is entered into the symbol table. This entry is flagged to indicate that the symbol is undefined.
- The address of the operand field of the instruction that refers to the undefined symbol is added to a list of forward references associated with the symbol table entry.
- When the definition for a symbol is encountered, the forward reference list for that symbol is scanned (if one exists), and the proper address is inserted into any instructions previously generated.

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

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

Symbol	Value
LENGTH	100C
RDREC	* → 2013 ∅
THREE	1003
ZERO	1006
WRREC	* → 201F ∅
EOF	1000
ENDFIL	* → 201C ∅
RETADR	1009
BUFFER	100F
CLOOP	2012
FIRST	200F

- Figure 2.19(a) shows the object code and symbol table entries as they would be after scanning line 40 of the program in fig. 2.18.
- The first forward reference occurred on line 15.
- Since the operand (**RDREC**) was not yet defined, the instruction was assembled with no value assigned as the operand address (denoted in the figure by **----).**
- **RDREC** was then entered into **SYMTAB** as an undefined symbol (indicated by *****); the address of the operand field of the instruction (2013) was inserted in a list associated with **RDREC**.
- A similar process was followed with the instructions on lines 30 and 35.

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

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 0
EOF	1000
ENDFIL	2024
RETADR	1009
BUFFER	100F
CLOOP	2012
FIRST	200F
MAXLEN	203A
INPUT	2039
EXIT	* → 2050 0
RLOOP	2043

- Now consider Fig. 2.19(b), which corresponds to the situation after scanning line 160.
- Some of the forward references have been resolved by this time, while others have been added.
- When the symbol **ENDFIL** was defined (line 45), the assembler placed its value in the **SYMTAB** entry; it then inserted this value into the instruction operand field (at address 201C) as directed by the forward reference list.
- From this point on, any references to **ENDFIL** would not be forward references, and would not be entered into a list.
- Similarly, the definition of **RDREC** (line 125) resulted in the filling in of the operand address at location 2013.
- Meanwhile, two new forward references have been added: to **WRREC** (line 65) and **EXIT** (line 155).
- At the end of the program, any **SYMTAB** entries that are still marked with * indicate undefined symbols.
- These should be flagged by the assembler as errors.
- When the end of the program is encountered, the assembly is complete.
- If no errors have occurred, the assembler searches **SYMTAB** for the value of the symbol named in the **END** statement (in this case, **FIRST**) and jumps to this location to begin execution of the assembled program.

- One-pass assemblers that produce object programs as output are often used on systems where external working-storage devices (for the intermediate file between the two passes) are not available.
- Such assemblers may also be useful when the external storage is slow or is inconvenient to use for some other reason.
- One-pass assemblers that produce object programs follow a slightly different procedure from that previously described.
- Forward references are entered into lists as before.
- Now, however, when the definition of a symbol is encountered, instructions that made forward references to that symbol may no longer be available in memory for modification.
- In general, they will already have been written out as part of a Text record in the object program.
- In this case 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 the action of the loader.

- Figure 2.20 illustrates this process.
- The second Text record contains the object code generated from lines 10 through 40 in Fig. 2.18.
- The operand addresses for the instructions on lines 15, 30, and 35 have been generated as 0000.
- When the definition of **ENDFIL** on line 45 is encountered, the assembler generates the third Text record.
- This record specifies that the value 2024 (the address of **ENDFIL**) is to be loaded at location 201C (the operand address field of the **JEQ** instruction on line 30).
- When the program is loaded, therefore, the value 2024 will replace the 0000 previously loaded.
- The other forward references in the program are handled in exactly the same way.
- In effect, the services of the loader are being used to complete forward references that could not be handled by the assembler.
- Of course, the object program records must be kept in their original order when they are presented to the loader.

Object program from one-pass assembler for program in Fig. 2.18 (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
```

Algorithm for One Pass Assembler (2.19 c)

- **See additional pdf**

Multi-Pass Assemblers

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

ALPHA	**EQU**	**BETA**
BETA	**EQU**	**DELTA**
DELTA	**RESW**	**1**

- The symbol **BETA** cannot be assigned a value when it is encountered during the first pass because **DELTA** has not yet been defined. As a result, **ALPHA** cannot be evaluated during the second pass.
- This means that any assembler that makes only two sequential passes over the source program cannot resolve such a sequence of definitions.
- As a matter of fact, such forward references tend to create difficulty for a person reading the program as well as for the assembler.
- The general solution is a multi-pass assembler that can make as many passes as are needed to process the definitions of symbols.
- It is not necessary for such an assembler to make more than two passes over the entire program.
- Instead, the portions of the program that involve forward references in symbol definition are saved during Pass 1.
- Additional passes through these stored definitions are made as the assembly progresses. This process is followed by a normal Pass 2.

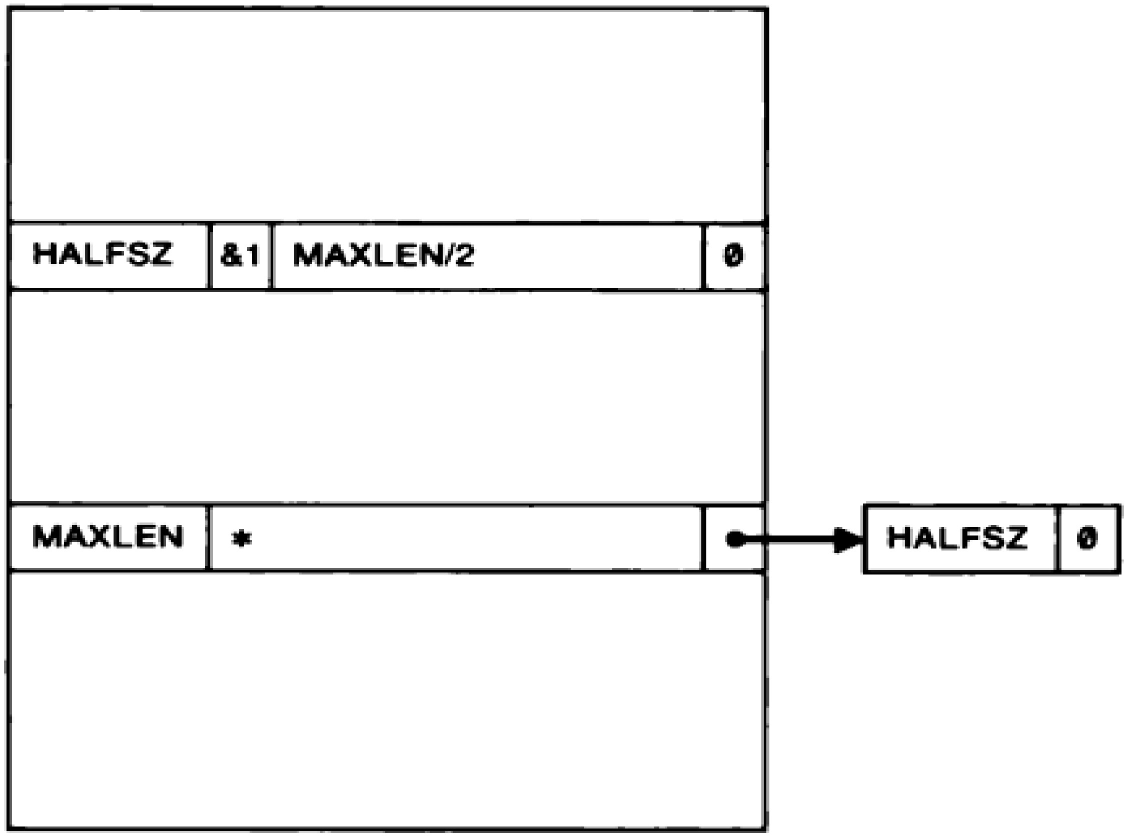
Example of Multi – Pass Assembler Operation (2.21 a)

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

- Figure 2.21(a) shows a sequence of symbol-defining statements that involve forward references; the other parts of the source program are not important for our discussion, and have been omitted.
- The following parts of Fig. 2.21 show information in the symbol table as it might appear after processing each of the source statements shown.

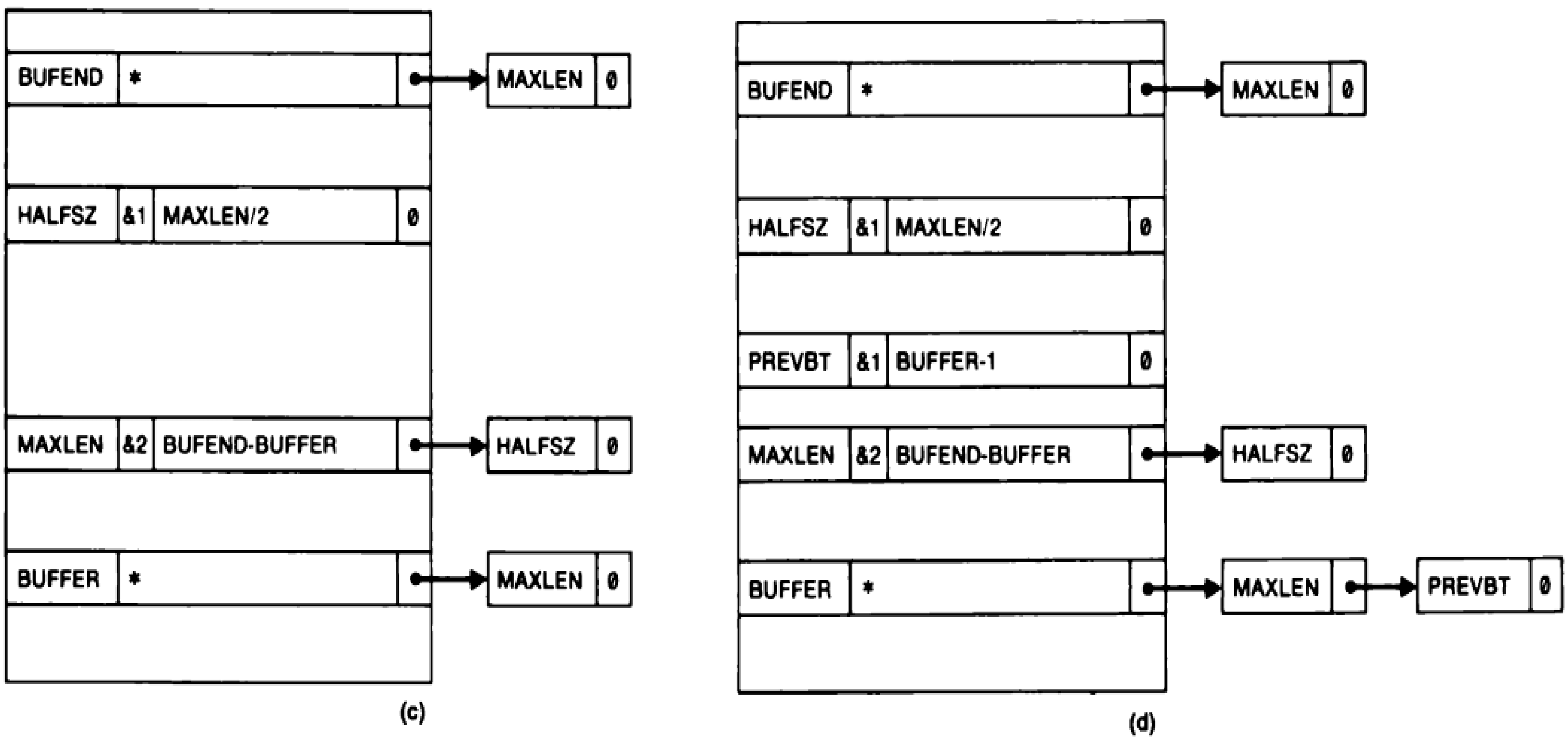
- Figure 2.21(b) displays symbol table entries resulting from Pass 1 processing of the statement: **HALFSZ EQU** ***MAXLEN/2***
- **MAXLEN** has not yet been defined, so no value for **HALFSZ** can be computed.
- The defining expression for **HALFSZ** is stored in the symbol table in place of its value.
- The entry **&1** indicates that one symbol in the defining expression is undefined.
- In an actual implementation, of course, this definition might be stored at some other location. **SYMTAB** would then simply contain a pointer to the defining expression.
- The symbol **MAXLEN** is also entered in the symbol table, with the flag ***** identifying it as undefined.
- Associated with this entry is a list of the symbols whose values depend on **MAXLEN** (in this case, **HALFSZ**).

(2.21 b)



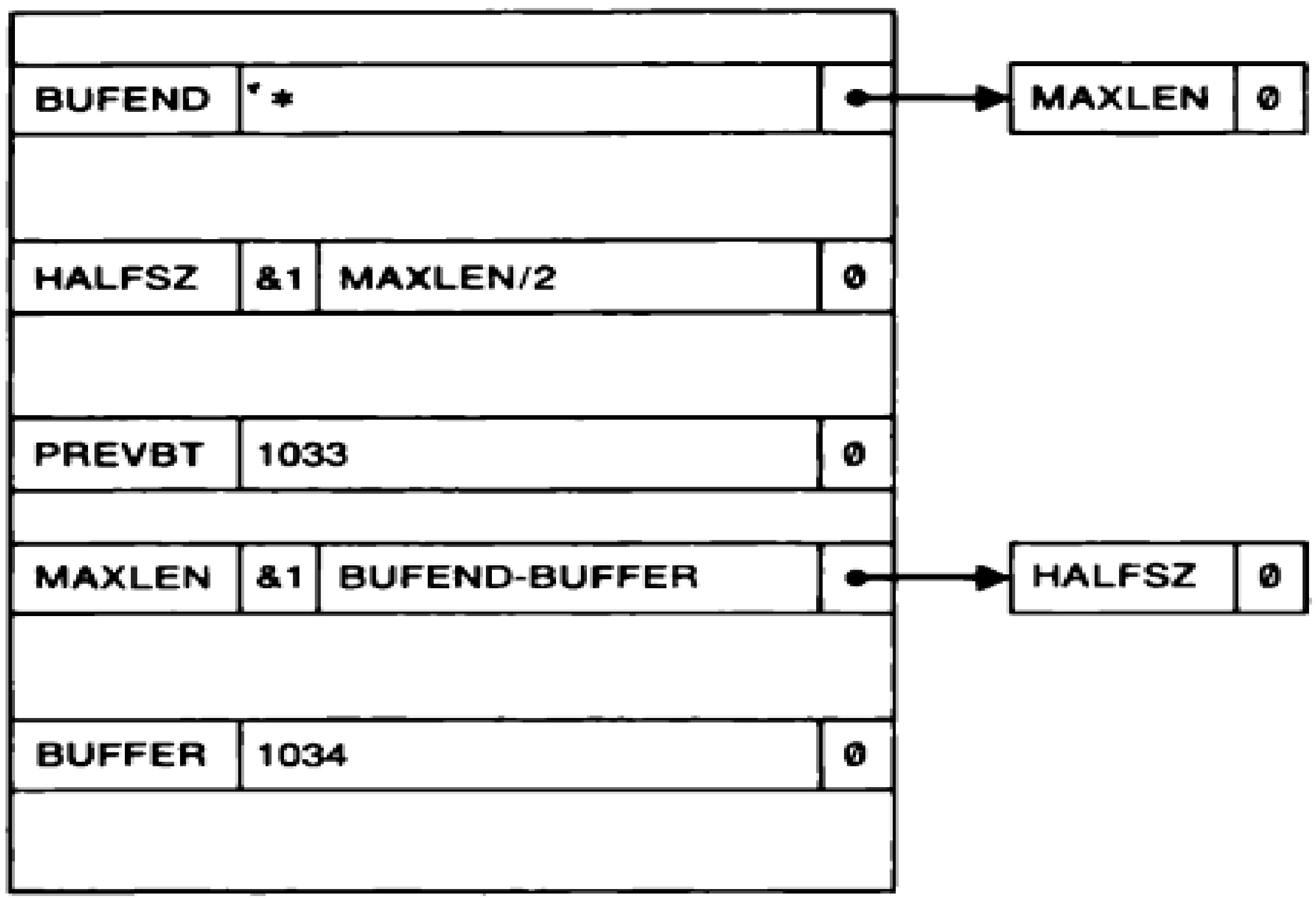
- The same procedure is followed with the definition of **MAXLEN** [see Fig. 2.21(c)].
- In this case there are two undefined symbols involved in the definition: **BUFEND** and **BUFFER**.
- Both of these are entered into **SYMTAB** with lists indicating the dependence of **MAXLEN** upon them.
- Similarly, the definition of **PREVBT** causes this symbol to be added to the list of dependencies on **BUFFER** [as shown in Fig. 2.21(d)].

(2.21 c and d)


BUFEND
*
MAXLEN
0
HALFSZ
&1
MAXLEN/2
0
MAXLEN
&2
BUFEND-BUFFER
HALFSZ
0
BUFFER
*
MAXLEN
0
(c)
BUFEND
*
MAXLEN
0
HALFSZ
&1
MAXLEN/2
0
PREVBT
&1
BUFFER-1
0
MAXLEN
&2
BUFEND-BUFFER
HALFSZ
0
BUFFER
*
MAXLEN
PREVBT
0
(d)

- So far we have simply been saving symbol definitions for later processing.
- The definition of **BUFFER** on line 4 lets us begin evaluation of some of these symbols.
- Let us assume that when line 4 is read, the location counter contains the hexadecimal value 1034. This address is stored as the value of **BUFFER**.
- The assembler then examines the list of symbols that are dependent on **BUFFER**.
- The symbol table entry for the first symbol in this list (**MAXLEN**) shows that it depends on two currently undefined symbols; therefore, **MAXLEN** cannot be evaluated immediately.
- Instead, the **&2** is changed to **&1** to show that only one symbol in the definition (**BUFEND**) remains undefined.
- The other symbol in the list (**PREVBT**) can be evaluated because it depends only on **BUFFER**.
- The value of the defining expression for **PREVBT** is calculated and stored in **SYMTAB**. The result is shown in Fig. 2.21(e).

(2.21 e)



BUFEND
*
MAXLEN
0
HALFSZ
&1
MAXLEN/2
0
PREVBT
1033
0
MAXLEN
&1
BUFEND-BUFFER
HALFSZ
0
BUFFER
1034
0

(e)

- The remainder of the processing follows the same pattern.
- When **BUFEND** is defined by line 5, its value is entered into the symbol table.
- The list associated with **BUFEND** then directs the assembler to evaluate **MAXLEN**, and entering a value for **MAXLEN** causes the evaluation of the symbol in its list (**HALFSZ**).
- As shown in Fig. 2.21(f), this completes the symbol definition process.
- If any symbols remained undefined at the end of the program, the assembler would flag them as errors.
- The procedure we have just described applies to symbols defined by assembler directives like **EQU**.

(2.21 f)

BUFEND	2034	Ø
HALFSZ	800	Ø
PREVBT	1033	Ø
MAXLEN	1000	Ø
BUFFER	1034	Ø

(f)