does not continue to the second pass. In this case, the assembly listing contains only errors that could be detected during Pass 1.

If no errors are detected during the first pass, the assembler proceeds to Pass 2. The second pass reads the source program again, instead of using an intermediate file as we discussed for SIC. This means that location counter values must be recalculated during Pass 2. It also means that any warning messages that were generated during Pass 1 (but were not serious enough to terminate the assembly) are lost. The assembly listing will contain only errors and warnings that are generated during Pass 2.

Assembled control sections are placed into the object program according to their storage mapping class. Executable instructions, read-only data, and various kinds of debugging tables are assigned to an object program section named .TEXT. Read/write data and TOC entries are assigned to an object program section named .DATA. Uninitialized data is assigned to a section named .BSS. When the object program is generated, the assembler first writes all of the .TEXT control sections, followed by all of the .DATA control sections except for the TOC. The TOC is written after the other .DATA control sections. Relocation and linking operations are specified by entries in a relocation table, similar to the Modification records we discussed for SIC.

EXERCISES

Section 2.1

1. Apply the algorithm described in Fig. 2.4 to assemble the source program in Fig. 2.1. Your results should be the same as those shown in Figs. 2.2 and 2.3.

2. Apply the algorithm described in Fig. 2.4 to assemble the following SIC source program:

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

3. As mentioned in the text, a number of operations in the algorithm of Fig. 2.4 are not explicitly spelled out. (One example would be scanning the instruction operand field for the modifier ",X".) List as many of these implied operations as you can, and think about how they might be implemented.

4. Suppose that you are to write a "disassembler"—that is, a system program that takes an ordinary object program as input and produces a listing of the source version of the program. What tables and data structures would be required, and how would they be used? How many passes would be needed? What problems would arise in recreating the source program?

5. Many assemblers use free-format input. Labels must start in Column 1 of the source statement, but other fields (opcode, operands, comments) may begin in any column. The various fields are separated by blanks. How could our assembler logic be modified to allow this?

6. The algorithm in Fig. 2.4 provides for the detection of some assembly errors; however, there are many more such errors that might occur. List error conditions that might arise during the assembly of a SIC program. When and how would each type of error be detected, and what action should the assembler take for each?

7. Suppose that the SIC assembler language is changed to include a new form of the RESB statement, such as

```
        RESB   n'c'
```

   which reserves *n* bytes of memory and initializes all of these bytes to the character 'c'. For example, line 105 in Fig. 2.5 could be changed to

```
BUFFER     RESB   4096' '
```

   This feature could be implemented by simply generating the required number of bytes in Text records. However, this could lead to a large increase in the size of the object program—for example, the object program in Fig. 2.8 would be about 40 times its previous size. Propose a way to implement this new form of RESB without such a large increase in object program size.

8. Suppose that you have a two-pass assembler that is written according to the algorithm in Fig. 2.4. In the case of a duplicate symbol,

this assembler would give an error message only for the second (i.e., duplicate) definition. For example, it would give an error message only for line 5 of the program below.

```
1       P3        START     1000
2                 LDA       ALPHA
.                 .         .
3                 STA       ALPHA
.                 .         .
4       ALPHA     RESW      1
.                 .         .
5       ALPHA     WORD      0
6                 END
```

Suppose that you want to change the assembler to give error messages for all definitions of a doubly defined symbol (e.g., lines 4 and 5), and also for all references to a doubly defined symbol (e.g., lines 2 and 3). Describe the changes you would make to accomplish this. In making this modification, you should change the existing assembler as little as possible.

9. Suppose that you have a two-pass assembler that is written according to the algorithm in Fig. 2.4. You want to change this assembler so that it gives a warning message for labels that are not referenced in the program, as illustrated by the following example.

```
P3        START     1000
          LDA       DELTA
          ADD       BETA
LOOP      STA       DELTA
Warning: label is never referenced
          RSUB
ALPHA     RESW      1
Warning: label is never referenced
BETA      RESW      1
DELTA     RESW      1
          END
```

The warning messages should appear in the assembly listing directly below the line that contains the unreferenced label, as shown above. Describe the changes you would make in the assembler to add this

new diagnostic feature. In making this modification, you should change the existing assembler as little as possible.

Section 2.2

1. Could the assembler decide for itself which instructions need to be assembled using extended format? (This would avoid the necessity for the programmer to code + in such instructions.)

2. As we have described it, the BASE statement simply gives information to the assembler. The programmer must also write an instruction like LDB to load the correct value into the base register. Could the assembler automatically generate the LDB instruction from the BASE statement? If so, what would be the advantages and disadvantages of doing this?

3. Generate the object code for each statement in the following SIC/XE program:

```
SUM         START    0
FIRST       LDX      #0
            LDA      #0
            +LDB     #TABLE2
            BASE     TABLE2
LOOP        ADD      TABLE,X
            ADD      TABLE2,X
            TIX      COUNT
            JLT      LOOP
            +STA     TOTAL
            RSUB
COUNT       RESW     1
TABLE       RESW     2000
TABLE2      RESW     2000
TOTAL       RESW     1
            END      FIRST
```

4. Generate the complete object program for the source program given in Exercise 3.

5. Modify the algorithm described in Fig. 2.4 to handle all of the SIC/XE addressing modes discussed. How would these modifications be reflected in the assembler designs discussed in Chapter 8?

6. Modify the algorithm described in Fig. 2.4 to handle relocatable programs. How would these modifications be reflected in the assembler designs discussed in Chapter 8?

7. Suppose that you are writing a disassembler for SIC/XE (see Exercise 2.1.4.) How would your disassembler deal with the various addressing modes and instruction formats?

8. Our discussion of SIC/XE Format 4 instructions specified that the 20-bit "address" field should contain the actual target address, and that addressing mode bits *b* and *p* should be set to 0. (That is, the instruction should contain a direct address—it should not use base relative or program-counter relative addressing.)

   However, it would be possible to use program-counter relative addressing with Format 4. In that case, the "address" field would actually contain a displacement, and bit *p* would be set to 1. For example, the instruction on line 15 in Fig. 2.6 could be assembled as

   ```
   0006  CLOOP  +JSUB   RDREC        4B30102C
   ```

   (using program-counter relative addressing with displacement 102C).

   What would be the advantages (if any) of assembling Format 4 instructions in this way? What would be the disadvantages (if any)? Are there any situations in which it would *not* be possible to assemble a Format 4 instruction using program-counter relative addressing?

9. Our Modification record format is well suited for SIC/XE programs because all address fields in instructions and data words fall neatly into half-bytes. What sort of Modification record could we use if this were not the case (that is, if address fields could begin anywhere within a byte and could be of any length)?

10. Suppose that we made the program in Fig. 2.1 a relocatable program. This program is written for the *standard* version of SIC, so all operand addresses are actual addresses, and there is only one instruction format. Nearly every instruction in the object program would need to have its operand address modified at load time. This would mean a large number of Modification records (more than doubling the size of the object program). How could we include the required relocation information without this large increase in object program size?

11. Suppose that you are writing an assembler for a machine that has *only* program-counter relative addressing. (That is, there are no direct-addressing instruction formats and no base relative addressing.) Suppose that you wish to assemble an instruction whose operand is an absolute address in memory—for example,

```
        LDA  100
```

to load register A from address (hexadecimal) 100 in memory. How might such an instruction be assembled in a relocatable program? What relocation operations would be required?

12. Suppose that you are writing an assembler for a machine on which the length of an assembled instruction depends upon the type of the operand. Consider, for example, the following three fragments of code:

a.
```
        ADD  ALPHA
        .
        .
ALPHA   DC   I(3)
```

b.
```
        ADD  ALPHA
        .
        .
ALPHA   DC   F(3.1)
```

c.
```
        ADD  ALPHA
        .
        .
ALPHA   DC   D(3.14159)
```

In case (a), ALPHA is an integer operand; the ADD instruction generates 2 bytes of object code. In case (b), ALPHA is a single-precision floating-point operand; the ADD instruction generates 3 bytes of object code. In case (c), ALPHA is a double-precision floating-point operand; the ADD instruction generates 4 bytes of object code.

What special problems does such a machine present for an assembler? Briefly describe how you would solve these problems—that is, how your assembler for this machine would be different from the assembler structure described in Section 2.1.

Section 2.3

1. Modify the algorithm described in Fig. 2.4 to handle literals.
2. In the program of Fig. 2.9, could we have used literals on lines 135 and 145? Why might we prefer *not* to use a literal here?
3. With a minor extension to our literal notation, we could write the instruction on line 55 of Fig. 2.9 as

   ```
   LDA    =W'3'
   ```

   specifying as the literal operand a word with the value 3. Would this be a good idea?
4. Immediate operands and literals are both ways of specifying an operand value in a source statement. What are the advantages and disadvantages of each? When might each be preferable to the other?
5. Suppose that you have a two-pass SIC/XE assembler that does not support literals. Now you want to modify the assembler to handle literals. However, you want to place the literal pool at the *beginning* of the assembled program, not at the end as is commonly done. (You do not have to worry about LTORG statements—your assembler should always place all literals in a pool at the beginning of the program.) Describe how you could accomplish this. If possible, you should do so without adding another pass to the assembler. Be sure to describe any data structures that you may need, and explain how they are used in the assembler.
6. Suppose we made the following changes to the program in Fig. 2.9:
   a. Delete the LTORG statement on line 93.
   b. Change the statement on line 45 to +LDA... .
   c. Change the operands on lines 135 and 145 to use literals (and delete line 185).

   Show the resulting object code for lines 45, 135, 145, 215, and 230. Also show the literal pool with addresses and data values. Note: you do not need to retranslate the entire program to do this.
7. Assume that the symbols ALPHA and BETA are labels in a source program. What is the difference between the following two sequences of statements?

a.
```
LDA   ALPHA-BETA
```

b.
```
LDA   ALPHA
SUB   BETA
```

8. What is the difference between the following sequences of statements?

a.
```
LDA   #3
```

b.
```
THREE  EQU   3
       .
       .
       LDA   #THREE
```

c.
```
THREE  EQU   3
       .
       .
       LDA   THREE
```

9. Modify the algorithm described in Fig. 2.4 to handle multiple program blocks.

10. Modify the algorithm described in Fig. 2.4 to handle multiple control sections.

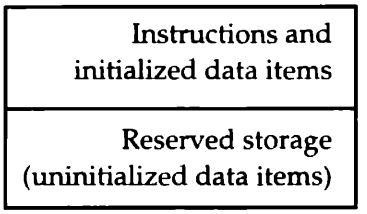
11. Suppose all the features we described in Section 2.3 were to be implemented in an assembler. How would the symbol table required be different from the one discussed in Section 2.1?

12. Which of the features described in Section 2.3 would create additional problems in the writing of a disassembler (see Exercise 2.1.4)? Describe these problems, and discuss possible solutions.

13. When different control sections are assembled together, some references between them could be handled by the assembler (instead of being passed on to the loader). In the program of Fig. 2.15, for example, the expression on line 190 could be evaluated directly by the assembler because its symbol table contains all of the required information. What would be the advantages and disadvantages of doing this?

14. In the program of Fig. 2.11, suppose we used only two program blocks: the default block and CBLKS. Assume that the data items in CDATA are to be included in the default block. What changes in the source program would accomplish this? Show the object program (corresponding to Fig. 2.13) that would result.

15. Suppose that for some reason it is desirable to separate the parts of an assembler language program that require initialization (e.g., instructions and data items defined with WORD or BYTE) from the parts that do not require initialization (e.g., storage reserved with RESW or RESB). Thus, when the program is loaded for execution it should look like



Suppose that it is considered too restrictive to require the programmer to perform this separation. Instead, the assembler should take the source program statements in whatever order they are written, and automatically perform the rearrangement as described above.

Describe a way in which this separation of the program could be accomplished by a two-pass assembler.

16. Suppose LENGTH is defined as in the program of Fig. 2.9. What would be the difference between the following sequences of statements?

a.
```
LDA     LENGTH
SUB     #1
```

b.
```
LDA     LENGTH-1
```

17. Referring to the definitions of symbols in Fig. 2.10, give the value, type, and intuitive meaning (if any) of each of the following expressions:

a. `BUFFER-FIRST`

b. `BUFFER+4095`

c. `MAXLEN-1`

d. `BUFFER+MAXLEN-1`

e. `BUFFER-MAXLEN`

f. `2*LENGTH`

g. `2*MAXLEN-1`

h. `MAXLEN-BUFFER`

i. `FIRST+BUFFER`

j. `FIRST-BUFFER+BUFEND`

18. In the program of Fig. 2.9, what is the advantage of writing (on line 107)

```
MAXLEN   EQU    BUFEND-BUFFER
```

instead of

```
MAXLEN   EQU    4096  ?
```

19. In the program of Fig. 2.15, could we change line 190 to

```
MAXLEN   EQU    BUFEND-BUFFER
```

and line 133 to

```
        +LDT    #MAXLEN
```

as we did in Fig. 2.9?

20. The assembler could simply assume that any reference to a symbol not defined within a control section is an external reference. This change would eliminate the need for the EXTREF statement. Would this be a good idea?

21. How could an assembler that allows external references avoid the need for an EXTDEF statement? What would be the advantages and disadvantages of doing this?

22. The assembler could automatically use extended format for instructions whose operands involve external references. This would eliminate the need for the programmer to code + in such statements. What would be the advantages and disadvantages of doing this?

23. On some systems, control sections can be composed of several different parts, just as program blocks can. What problems does this pose for the assembler? How might these problems be solved?

24. Assume that the symbols RDREC and COPY are defined as in Fig. 2.15. According to our rules, the expression

```
RDREC-COPY
```

would be illegal (that is, the assembler and/or the loader would reject it). Suppose that for some reason the program really needs the value of this expression. How could such a thing be accomplished without changing the rules for expressions?

25. We discussed a large number of assembler directives, and many more could be implemented in an actual assembler. Checking for them one at a time using comparisons might be quite inefficient. How could we use a table, perhaps similar to OPTAB, to speed recognition and handling of assembler directives? (Hint: the answer to this problem may depend upon the language in which the assembler itself is written.)

26. Other than the listing of the source program with generated object code, what assembler outputs might be useful to the programmer? Suggest some optional listings that might be generated and discuss any data structures or algorithms involved in producing them.

Section 2.4

1. The process of fixing up a few forward references should involve less overhead than making a complete second pass of the source program. Why don't all assemblers use the one-pass technique for efficiency?

2. Suppose we wanted our assembler to produce a cross-reference listing for all symbols used in the program. For the program of Fig. 2.5, such a listing might look like

Symbol	Defined on line	Used on lines
COPY	5	
FIRST	10	255
CLOOP	15	40
ENDFIL	45	30
EOF	80	45
RETADR	95	10,70
LENGTH	100	12,13,20,60,175,212

How might this be done by the assembler? Indicate changes to the logic and tables discussed in Section 2.1 that would be required.

3. Could a one-pass assembler produce a relocatable object program and handle external references? Describe the processing logic that would be involved and identify any potential difficulties.
4. How could literals be implemented in a one-pass assembler?
5. We discussed one-pass assemblers as though instruction operands could only be single symbols. How could a one-pass assembler handle an instruction like

```
JEQ     ENDFIL+3
```

where ENDFIL has not yet been defined?

6. Outline the logic flow for a simple one-pass load-and-go assembler.
7. Using the methods outlined in Chapter 8, develop a modular design for a one-pass assembler that produces object code in memory.
8. Suppose that an instruction involving a forward reference is to be assembled using program-counter relative addressing. How might this be handled by a one-pass assembler?
9. The process of fixing up forward references in a one-pass assembler that produces an object program is very similar to the linking process described in Section 2.3.5. Why didn't we just use Modification records to fix up the forward references?
10. How could we extend the methods of Section 2.4.2 to handle forward references in ORG statements?

Section 2.5

1. Consider the description of the VAX architecture in Section 1.4.1. What characteristics would you expect to find in a VAX assembler?
2. Consider the description of the T3E architecture in Section 1.5.3. What characteristics would you expect to find in a T3E assembler?