Chapter 2

ASSEMBLY

2.1 ASSEMBLERS AND RELATED PROGRAMS

The simplest assembler program is the *load-and-go* assembler. It accepts as input a program whose instructions are essentially in one-to-one correspondence with those of machine language, but with symbolic names used for operations and operands. It produces as output a machine-language program in main storage, ready to be executed. The translation is usually performed in a single pass over the input program text. The resulting machine-language program occupies storage locations which are fixed at the time of translation and cannot be changed subsequently. The program can call library subroutines, provided that they occupy other locations than those required by the program. No provision is made for combining separate subprograms translated in this manner.

The load-and-go assembler forgoes the advantages of modular program development. Among the most important of these are (1) the ability to design, code, and test different program components in parallel; and (2) the restriction of changes to only applicable modules rather than throughout the program. Because some program development costs rise faster than proportionally to the length of a program component, another benefit of modularization is reduction of these costs. Most assemblers are therefore designed to satisfy the desire to create programs in modules. These *module* assemblers, also called "routine" or "subprogram" assemblers (cf. Barron [1972]), generally embody a two-pass translation. During the first pass the assembler examines the assembler-language program and collects the symbolic names into a table. During the second pass, the assembler generates code which is not quite in machine language. It is rather in a similar form, sometimes called "relocatable code" and here called *object code*. The program module in object-code form is typically called an *object module*.

The assembler-language program contains three kinds of entities. *Absolute* entities include operation codes, numeric and string-valued constants,

and fixed addresses. The values of absolute entities are independent of which storage locations the resulting machine code will eventually occupy. *Relative* entities include the addresses of instructions and of working storage. These are fixed only with respect to each other, and are normally stated relative to the address of the beginning of the module. An *externally defined* entity is used within a module but not defined within it. Whether it is in fact absolute or relative is not necessarily known at the time the module is translated.

The object module includes identification of which addresses are relative, which symbols are defined externally, and which internally defined symbols are expected to be referenced externally. In the modules in which the latter are used, they are considered to be externally defined. These external references are resolved for two or more object modules by a linker. The linker accepts the several object modules as input and produces a single module ready for loading, hence termed a *load module.*

The load module is free* of external references and consists essentially of machine-language code accompanied by a specification of which addresses are relative. When the actual main storage locations to be occupied by the program become known, a relocating loader reads the program into storage and adjusts the relative addresses to refer to those actual locations. The output from the loader is a machine-language program ready for execution. The over-all process is depicted in Fig. 2.1.

If only a single source-language module containing no external references is translated, it can be loaded directly without intervention by the linker. In some programming systems the format of linker output is sufficiently compatible with that of its input to permit the linking of a previously produced load module with some new object modules.

The functions of linking and loading are sometimes both effected by a single program, called a *linking loader*. Despite the convenience of combining the linking and loading functions, it is important to realize that they are distinct functions, each of which can be performed independently of the other.

In this chapter we examine in some detail the function and implementation of those assemblers which provide neither macro processing nor conditional assembly. Those functions are presented in Chapters 4 and 5, respectively. Because of its importance, the standard two-pass assembler is presented in detail, followed by a briefer description of two one-pass assemblers. Before describing a full-function two-pass assembler, however,

*This is an oversimplification. The module may still contain so-called *weak* external references which will actually not be made during the ensuing execution.

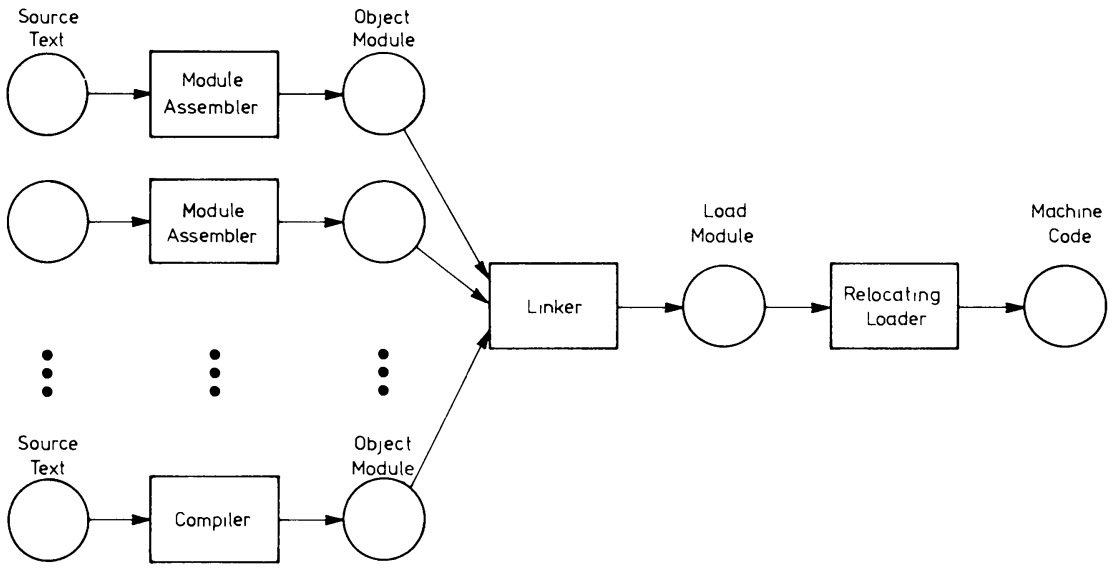

Figure 2.1 Program Translation Sequence

we concentrate on basic aspects of assembly by examining first a rudimentary assembler.

2.2 RUDIMENTARY TWO-PASS ASSEMBLER

2.2.1 *Function*

The program of Fig. 2.2, although written in a hypothetical assembler language for a mythical computer, contains the basic elements which need to be translated into machine language. For ease of reference, each instruction is identified by a line number, which is *not* part of the program. Each instruction in our language contains either an operation specification (lines 1-15) or a storage specification (lines 16-21). An operation specification is a symbolic operation code, which may be preceded by a label and must be followed by 0, 1, or 2 operand specifications, as appropriate to the operation. A storage specification is a symbolic instruction to the assembler. In our assembler language, it must be preceded by a label and must be followed, if appropriate, by a constant. Labels and operand specifications are symbolic addresses; every operand specification must appear somewhere in the program as a label.

Our machine has a single accumulator and a main storage of unspecified size. Its 14 instructions are listed in Fig. 2.3. The first column shows the assembler-language operation code and the second gives the machine-language equivalent (in decimal). The fourth column specifies the number

Line	Label	Operation	Operand 1	Operand 2
1		COPY	ZERO	OLDER
2		COPY	ONE	OLD
3		READ	LIMIT	
4		WRITE	OLD	
5	FRONT	LOAD	OLDER	
6		ADD	OLD	
7		STORE	NEW	
8		SUB	LIMIT	
9		BRPOS	FINAL	
10		WRITE	NEW	
11		COPY	OLD	OLDER
12		COPY	NEW	OLD
13		BR	FRONT	
14	FINAL	WRITE	LIMIT	
15		STOP		
16	ZERO	CONST	0	
17	ONE	CONST	1	
18	OLDER	SPACE		
19	OLD	SPACE		
20	NEW	SPACE		
21	LIMIT	SPACE		

Figure 2.2 Sample Assembler-Language Program

Operation code			No. of	
Symbolic	Machine	Length	operands	Action
ADD	02	2	1	ACC ← ACC + OPD1
BR	00	2	1	branch to OPD1
BRNEG	05	2	1	branch to OPD1 if ACC < 0
BRPOS	01	2	1	branch to OPD1 if ACC > 0
BRZERO	04	2	1	branch to OPD1 if ACC = 0
COPY	13	3	2	OPD2 ← OPD1
DIVIDE	10	2	1	ACC ← ACC ÷ OPD1
LOAD	03	2	1	ACC ← OPD1
MULT	14	2	1	ACC ← ACC × OPD1
READ	12	2	1	OPD1 ← input stream
STOP	11	1	0	stop execution
STORE	07	2	1	OPD1 ← ACC
SUB	06	2	1	ACC ← ACC − OPD1
WRITE	08	2	1	output stream ← OPD1

Figure 2.3 Instruction Set

of operands, and the last column describes the action which ensues when the instruction is executed. In that column "ACC", "OPD1", and "OPD2" refer to contents of the accumulator, of the first operand location, and of the second operand location, respectively. The length of each instruction in words is 1 greater than the number of its operands. Thus if the machine has 12-bit words, an ADD instruction is 2 words, or 24 bits, long. The table's third column, which is redundant, gives the instruction length. If our mythical computer had a fixed instruction length, the third and fourth columns could both be omitted.

The storage specification SPACE reserves one word of storage which presumably will eventually hold a number; there is no operand. The storage specification CONST also reserves a word of storage; it has an operand which is the value of a number to be placed in that word by the assembler.

The instructions of the program are presented in four fields, and might indeed be constrained to such a format on the input medium. The label, if present, occupies the first field. The second field contains the symbolic operation code or storage specification, which will henceforth be referred to simply as the *operation*. The third and fourth fields hold the operand specifications, or simply *operands*, if present.

Although it is not important to our discussion to understand what the example program does, the foregoing specifications of the machine and of its assembler language reveal the algorithm. The program uses the recursion relation $f_i = f_{i-1} + f_{i-2}$, implemented by the assignment NEW ← OLD + OLDER, to compute the so-called *Fibonacci numbers* (0, 1, 1, 2, 3, 5, 8, . .). The program prints all numbers beyond the zero, but not exceeding the positive integer limit read in on line 3 of the program, and then prints that limit. The astute reader will observe that the program could be improved. For example, the loop could be rewritten to require only one branch. Nevertheless, the forward branch (line 9) is an important component of many longer assembler-language programs and it has been left in this short example on purpose.

Now that we have seen the elements of an assembler-language program, we can ask what functions the assembler must perform in translating it. Here is the list.

Replace symbolic addresses by numeric addresses.
Replace symbolic operation codes by machine operation codes.
Reserve storage for instructions and data.
Translate constants into machine representation.

The assignment of numeric addresses can be performed, except by a load-and-go assembler (which we are not now considering), without fore-

knowledge of what actual locations will eventually be occupied by the assembled program. It is necessary only to generate addresses relative to the start of the program. We shall assume that our assembler normally assigns addresses starting at 0. In translating line 1 of our example program, the resulting machine instruction will therefore be assigned address 0 and occupy 3 words, because COPY instructions are 3 words long. Hence the instruction corresponding to line 2 will be assigned address 3, the READ instruction will be assigned address 6, and the WRITE instruction of line 4 will be assigned address 8, and so on to the end of the program. But what addresses will be assigned to the operands named ZERO and OLDER? These addresses must be inserted in the machine-language representation of the first instruction.

2.2.2 *Implementation*

The assembler uses a counter to keep track of machine-language addresses. Because these addresses will ultimately specify locations in main storage, the counter is called the *location counter*, although "address counter" would be more accurate terminology. Before assembly, the location counter is initialized to zero. After each source line has been examined on the first pass, the location counter is incremented by the length of the machine-language code which will ultimately be generated to correspond to that source line.

When the assembler first encounters line 1 of the example program, it cannot replace the symbols ZERO and OLDER by addresses because those symbols make *forward references* to source-language program lines not yet reached by the assembler. The most straightforward way to cope with the problem of forward references is to examine the entire program text once, before attempting to complete the translation. During that examination, the assembler determines the address which corresponds to each symbol, and places both the symbols and their addresses in a *symbol table*. This is possible because each symbol used in an operand field must also appear as a label. The address corresponding to a label is just the address of the first word of the machine-language code for the line which that label begins. Those addresses are supplied by the location counter. Creation of the symbol table requires one pass over the source text. During a second pass, the assembler uses the addresses collected in the symbol table to perform the translation. As each symbolic address is encountered in the second pass, the corresponding numeric address is substituted for it in the object code.

Two of the most common logical errors in assembler-language pro-

gramming involve improper use of symbols. If a symbol appears in the operand field of some instruction, but nowhere in a label field, it is *undefined*. If a symbol appears in the label fields of more than one instruction, it is *multiply defined*. In building the symbol table on the first pass, the assembler must examine the label field of each instruction to permit it to associate the location counter value with each symbol. Multiply-defined symbols will be found on this pass. Undefined symbols, on the other hand, will not be found on the first pass unless the assembler also examines operand fields for symbols. Although this examination is not required for construction of the symbol table, normal practice is to perform it anyhow, because of its value in early detection of program errors.

There are many ways to organize a symbol table. Although the manner in which the symbol table is implemented is of considerable practical importance, it is not immediately relevant to the present discussion. We shall simply treat the symbol table as a set of (symbol,address) pairs, and assume the existence of procedures for insertion, search, and retrieval. In the illustrations which follow, the set of pairs is represented by a linear list in the order in which the symbols are first encountered. Symbol table organization is discussed in Section 6.4.

The state of processing after line 3 has been scanned is shown in Fig. 2.4. During processing of line 1, the symbols ZERO and OLDER were encountered and entered into the first two positions of the symbol table. The operation COPY was identified, and instruction length information from Fig. 2.3 used to advance the location counter from 0 to 3. During process-

Line	Address	Label	Operation	Operand 1	Operand 2
1	0		COPY	ZERO	OLDER
2	3		COPY	ONE	OLD
3	6		READ	LIMIT	

(a) Source text scanned

Symbol	Address
ZERO	---
OLDER	---
ONE	---
OLD	---
LIMIT	---

Location counter: 8

Line counter: 4

(b) Symbol table; Counters

Figure 2.4 First Pass After Scanning Line 3

ing of line 2 two more symbols were encountered and entered in the symbol table, and the location counter was advanced from 3 to 6. Line 3 yielded the fifth symbol, LIMIT, and caused incrementation of the location counter from 6 to 8. At this point the symbol table holds five symbols, none of which yet has an address. The location counter holds the address 8, and processing is ready to continue from line 4. Neither the line numbers nor the addresses shown in part (a) of the figure are actually part of the source-language program. The addresses record the history of incrementation of the location counter; the line numbers permit easy reference. Clearly, the assembler needs not only a location counter, but also a *line counter* to keep track of which source line is being processed.

During processing of line 4 the symbol OLD is encountered for the second time. Because it is already in the symbol table, it is not entered again. During processing of line 5, the symbol FRONT is encountered in the label field. It is entered into the symbol table, and the current location counter value, 10, is entered with it as its address. Figure 2.5 displays the state of the translation after line 9 has been processed.

Line	Address	Label	Operation	Operand 1	Operand 2
1	0		COPY	ZERO	OLDER
2	3		COPY	ONE	OLD
3	6		READ	LIMIT	
4	8		WRITE	OLD	
5	10	FRONT	LOAD	OLDER	
6	12		ADD	OLD	
7	14		STORE	NEW	
8	16		SUB	LIMIT	
9	18		BRPOS	FINAL	

(a) Source text scanned

Symbol	Address	
ZERO	---	
OLDER	---	
ONE	---	
OLD	---	
LIMIT	---	
FRONT	10	Location counter: 20
NEW	---	
FINAL	---	Line counter: 10

(b) Symbol table; Counters

Figure 2.5 First Pass after Scanning Line 9

The first pass continues in the same manner until line 14. There a label is encountered which is already in the symbol table, because it first appeared as an operand, on line 9. The symbol does not need to be entered again in the table, but its address, which has just become known, does. Each of lines 16–21 corresponds to one word of machine code, and the location counter is therefore incremented by 1 for each line. Each of these lines also includes a label, and the corresponding address for each is entered in the symbol table. At the end of the first pass, the state depicted in Fig. 2.6 has been reached.

The code generation is performed by our simple assembler during a second pass over the source text. Before starting the second pass, the line counter is reset to 1 and the location counter to 0. For each line of source code the assembler now produces a line of object code, which consists of the address, length, and text of the corresponding machine-language representation. The line and location counters are incremented as on the first pass. Line 1 is translated into 00 3 13 33 35 to indicate address 0, length 3 words, operation code 13, and operand addresses 33 and 35. The machine operation code is found in Fig. 2.3. This operation code table is fixed for all assemblies, being defined by the assembler and machine languages, and is part of the assembler program. The numeric addresses are found in the address fields of the symbol table entries for ZERO and OLDER. The symbol table is different, of course, for each assembly and is built during the first pass.

Successive lines are translated in the same manner. When line 5 is reached, a label is found. How is it treated on the second pass? It is ignored, because nothing in the machine code corresponds to the label field, and the address is given by the location counter. Thus the output corresponding to line 5 is 10 2 03 35. Lines 1–15, which contain operation specifications in the operation field, are all translated in this manner. Thus we see that whereas the first pass is concerned chiefly with advancing the location counter and building the symbol table, the second pass uses the symbol table to generate the object program.

But what about lines 16–21? For these the content of the operation field is a storage specification. The corresponding machine code is not an instruction. In fact, CONST specifies that its operand is to be placed in one word of machine code and SPACE specifies only that one word of machine code is to be reserved. Thus the object code produced from source line 16 is 33 1 00 and that corresponding to line 17 is 34 1 01. The content of the word corresponding to line 18 is not specified, and anything can be assembled. Using "XX" to represent an arbitrary value we can write the object code corresponding to line 18 as 35 1 XX. Figure 2.7 presents the object code which corresponds to the entire source-language program.

Line	Address	Label	Operation	Operand 1	Operand 2
1	0		COPY	ZERO	OLDER
2	3		COPY	ONE	OLD
3	6		READ	LIMIT	
4	8		WRITE	OLD	
5	10	FRONT	LOAD	OLDER	
6	12		ADD	OLD	
7	14		STORE	NEW	
8	16		SUB	LIMIT	
9	18		BRPOS	FINAL	
10	20		WRITE	NEW	
11	22		COPY	OLD	OLDER
12	25		COPY	NEW	OLD
13	28		BR	FRONT	
14	30	FINAL	WRITE	LIMIT	
15	32		STOP		
16	33	ZERO	CONST	0	
17	34	ONE	CONST	1	
18	35	OLDER	SPACE		
19	36	OLD	SPACE		
20	37	NEW	SPACE		
21	38	LIMIT	SPACE		

(a) Source text scanned

Symbol	Address	
ZERO	33	
OLDER	35	
ONE	34	
OLD	36	
LIMIT	38	
FRONT	10	Location counter: 39
NEW	37	
FINAL	30	Line counter: 22

(b) Symbol table; Counters

Figure 2.6 Result of First Pass

The XX can be thought of as a specification to the loader, which will eventually process the object code, that the content of the location corresponding to address 35 does not need to have any specific value loaded. The loader can then just skip over that location. Some assemblers specify anyway a particular value for reserved storage locations, often zeros. There

Address	Length	Machine code
00	3	13 33 35
03	3	13 34 36
06	2	12 38
08	2	08 36
10	2	03 35
12	2	02 36
14	2	07 37
16	2	06 38
18	2	01 30
20	2	08 37
22	3	13 36 35
25	3	13 37 36
28	2	00 10
30	2	08 38
32	1	11
33	1	00
34	1	01
35	1	XX
36	1	XX
37	1	XX
38	1	XX

Figure 2.7 Object Code Generated on Second Pass

is no logical requirement to do so, however, and the user unfamiliar with his assembler is ill-advised to count on a particular value.

The specifications CONST and SPACE do not correspond to machine instructions. They are really instructions to the assembler program. Because of this, we shall refer to them as *assembler instructions*. Another common designation for them is "pseudo-instructions". Neither term is really satisfactory. Of the two types of assembler instructions in our example program, one results in the generation of machine code and the other in the reservation of storage. Later we shall see assembler instructions which result in neither of these actions. The assembler instructions are given in a table available to the assembler. One organization is to use a separate table which is usually searched before the operation code table is searched. Another is to include both machine operations and assembler instructions in the same table. A field in the table entry then identifies the type to the assembler.

A few variations to the foregoing process can be considered. Some of the translation can actually be performed during the first pass. Operation fields must be examined during the first pass to determine their effect on

the location counter. The second-pass table lookup to determine the machine operation code can be obviated at the cost of producing intermediate text which holds machine operation code and instruction length in addition to source text. Another translation which can be performed during the first pass is that of constants, e.g. from source-language decimal to machine-language binary. The translation of any symbolic addresses which refer backward in the text, rather than forward, could be performed on the first pass, but it is more convenient to wait for the second pass and treat all symbolic addresses uniformly.

A minor variation is to assemble addresses relative to a starting address other than 0. The location counter is merely initialized to the desired address. If, for example, the value 200 is chosen, the symbol table would appear as in Fig. 2.8. The object code corresponding to line 1 would be 200 3 13 233 235.

Symbol	Address
ZERO	233
OLDER	235
ONE	234
OLD	236
LIMIT	238
FRONT	210
NEW	237
FINAL	230

Figure 2.8 Symbol Table with Starting Location 200

If it were known at assembly time that the program is to reside at location 200 for execution, then full object code with address and length need not be generated. The machine code alone would suffice. In this event the result of translation would be the following 39-word sequence.

```
 13  233  235   13  234  236   12  238   08  236   03  235   02
236   07  237   06  238   01  230   08  237   13  236  235   13
237  236   00  210   08  238   11   00   01   XX   XX   XX   XX
```

2.3 FULL TWO-PASS ASSEMBLER

2.3.1 *Functions*

Virtually all assembler languages incorporate more facilities for specifying machine function than does the rudimentary example presented in the previous section. We now examine the rich variety of functions which may

be present. In Section 2.3.2 we discuss how each can be implemented in an assembler.

Symbolic Instructions. As we have seen, the operation code may be represented symbolically. The symbol used is often called a *mnemonic operation code*, or simply a *mnemonic*, because it is usually selected for its mnemonic significance. Thus ADD and LOAD, or even A and L, are more likely to be used for instructions which add and load than are, say XB5 and ZARF. Symbolic representation can be extended to other fields of the instruction, such as indexing and indirect addressing specifications. One useful extension is the so-called *extended mnemonic*. This has the form of a symbolic operation code, but designates more information than just the machine operation code. The IBM 360, for example, has a general-purpose operation code for branch on condition (mnemonic BC) which is specialized to a particular condition by a 4-bit mask field. The extended mnemonic BZ (branch on zero) specifies both the 8-bit operation code 01000111 and the 4-bit mask 1000.

Symbolic Addresses. A symbol selected by the programmer is used to refer to the location which an item will occupy when the program is executed. For an item which occupies more than one addressable location, the address of the first location is normally used. For example, if a 4-byte integer-valued field named PRESSURE occupies storage locations known to the assembler as 1836–1839 in a byte-addressed machine, then the address 1836 would be associated with the symbol PRESSURE. A standard, but unfortunate, terminology is to refer to the address as the "value" of the symbol. In discussing an assembly, one would then say "the value of PRESSURE is 1836" to refer to the address of the symbol. This must be distinguished from a reference to the numeric value of the operand (say, 1013) stored at the address (1836) designated by the symbol PRESSURE. To avoid this confusion, and to simplify reference to values of other attributes of a symbol than its address, "value" will not be used here in the specialized sense of "address".

Symbolic addresses are not limited to single symbols. Arithmetic expressions involving one or more symbols are often permitted, and are extremely useful. Examples of such *address expressions* are FRONT + 5, 4 * OFFSET + I, and PRESSURE − RESERVE. More complicated address expressions are permitted by some assemblers.

Storage Reservation. In most assembler languages it is not necessary to place a label on each storage reservation or constant definition. The use of address expressions in the source language or of indexing in the machine operations provides easy access to unlabeled locations.

An assembler instruction to reserve storage may specify directly the

amount to be reserved. This is particularly common if the unit of allocation is fixed, as on most word-addressed machines. Thus one might have assembler codes RSF (reserve storage, first address) and RSL (reserve storage, last address) which require a label and one operand. The instruction ALPHA RSF 23 would reserve a block of 23 words and associate the symbol ALPHA with the first word of the block. The instruction OMEGA RSL 7 would reserve a block of 7 words, and associate the symbol OMEGA with the last word of the block.

If different units of allocation are used for different data types, it may be more convenient to specify quantities and units and rely on the assembler to compute the amount of storage required. The DS (define storage) operation of the IBM OS/360 assembler is of this type. The instruction WORK DS 15D,6F causes 144 bytes to be reserved (fifteen 8-byte double words and six 4-byte full words). The address of the label WORK is the address of the first byte of the 144.

Data Generation. Data to be placed in the object code are usually encoded in the assembler-language program in a form in which they are easily written. Although a machine may encode numbers in binary, the programmer is likely to think of their values in decimal. It is necessary to specify, either explicitly or implicitly, both the source and the target encodings.

Near one extreme might be a machine with two data types and a restricted assembler language. The only data are one-word numbers in either fixed-point or floating-point binary encoding. The assembler language permits decimal encodings only; floating-point is to be used if and only if a decimal point is present. In this situation a single assembler instruction suffices. For example,

```
RATES    CONST    8000,0.22,12000,0.25,16000,0.28
```

would reserve storage for 6 words, the first of them labeled RATES, and generate fixed- and floating-point values in them alternately.

If several encodings are possible, it may be more convenient to provide a distinct assembler instruction for each type of conversion. Thus DECIMAL and OCTAL might require operands in decimal and octal, respectively, and implicitly specify their conversion to binary. If there are many different types of conversions, the language may be more tractable if a single assembler instruction is used in conjunction with type indicators. Thus the OS/360 assembler instruction DC (define constant) permits 15 types of conversions. For example, DC F'5',H'−9',C'EXAMPLE' causes the generation of the integer 5 as a 4-byte full word, followed by the integer −9

as a 2-byte half word, followed by 7 bytes of characters. Repetition factors may be permitted, to assist in such specifications as that of a table all of whose values are initially zero.

Location Counters. The programmer is generally free to pick a nonzero origin for the location counter. An assembler instruction, perhaps called ORIGIN, has an operand to define the value of the location counter. In some assembler languages, this redefinition of the location counter can occur not only at the start of the source program, but at any point thereafter. That feature is useful chiefly with assemblers which provide two or more location counters.

Multiple location counters permit text to be permuted during translation. One example of a useful permutation is to write storage reservation and data generation specifications next to the associated source-language instructions, but to assemble data fields into a single area at the end of the program. Another is to interleave, in the source code, instructions to two or more processors (e.g. a machine and its I/O channels) but to segregate them in the object code.

Scope of Symbols. If two or more modules are to be combined after assembly, some symbolic addresses will be defined in one module and used in another. Most symbols, however, will be referenced only within the module in which they are defined. These symbols are said to be *local* to that module. A symbol defined within a module is called *global* if another module is to reference it. During the assembly of one module, the assembler can identify symbols used but not defined within the module. Each such use is either an error or a reference to a global variable whose definition must be supplied by another module during linking. The assembler cannot tell, however, whether a symbol which is defined in the module being translated will be referenced by another module. Yet this information must be transmitted by the assembler to the linker. Consequently, the assembler must itself be informed. One way is to mark the definition of each global symbol with a special tag. Thus

```
*RESULT   SPACE   1
```

might identify RESULT as external to the module, although internally defined. An alternative to the marker is an assembler instruction, say INTDEF. The sequence

```
RESULT    INTDEF
          SPACE   1
```

would have the same effect. The SPACE instruction can be given a label local to the module, as in

```
RESULT   INTDEF
ANSWER   SPACE  1
```

where the local symbol ANSWER can be used only within the module, whereas the global symbol RESULT can be used either within it or without. The advantage of providing a local label is that in some implementations the use of a global label may be more expensive. The alternative syntax

```
         INTDEF RESULT
         ---
RESULT   SPACE  1
```

does not require adjacency of the definition of the label to its designation as global. It does not allow a local label, however, and it suffers from the irregular use of the operand field for symbol definition rather than for symbol reference.

To permit independent assembly of more than one module during a single invocation of the assembler, it is necessary to distinguish the beginning and end of each assembler-language program. Even if only one assembly is to be performed, assembler instructions, often START and END, are provided to delimit the source program. These can be thought of as constituting vertical parentheses which define the scope of local symbols.

Redefinable Symbols. A symbol whose address depends upon its context is said to be *redefinable*. It cannot be held in the symbol table in the usual manner, because different occurrences may be associated with different addresses. The most frequently used redefinable symbol has as its address the current value of the location counter; often an asterisk is used. A common use of such a symbol is in relative branching. Thus BR $*-6$, wherever it appears in the program, is a branch to an address which is 6 less than the current address.

Another extremely useful type of redefinable symbol is also called a *local label*, but with a more restrictive connotation than merely "non-global". Distinguished syntactically from other symbols, it is defined afresh each time it appears in the label field of an instruction. Any use of such a symbol in an operand field refers to the most recent definition. A particularly convenient variation is to permit the reference to be marked either as backward to the most recent definition or as forward to the next definition. If, for example, normal labels are restricted to begin with an alphabetic character, the decimal digits might be reserved as ten local labels. Thus 3B would be a reference to the closest previous occurrence of local label 3; and 1F, to the next forthcoming occurrence of local label 1.

A local label (under this stronger definition) can be thought of as having a scope restricted to only a portion of the module in which it is defined.

Base Registers. The specification of main storage locations by address fields of machine-language instructions is commonly performed in one of three ways. (1) The address field is a single number which refers directly to the storage location. This is *direct* addressing. (2) The address field is a single number which specifies the displacement of the storage location from the origin of the program segment. Before execution, the origin is placed in a base register by the operating system. During interpretation of the instruction at execution time, the displacement is automatically added to the base register content to yield the storage location. Because the instruction does not refer explicitly to a base register, this is *implicit-base* addressing. (3) The address field is a pair of numbers, of which one designates a base register and the other constitutes the displacement. This is *explicit-base* addressing.

If the target machine employs direct addressing, any storage location is addressable. If it employs implicit-base addressing, addressability is a consequence of limiting the length of program segments to the number of locations that can be distinguished by the displacement. If, however, the target machine employs explicit-base addressing, two constraints impose a burden on the assembler in ensuring that all program locations will be addressable. One is that the displacement field is usually shorter than for implicit-base addressing, because some bits must be used to designate the base register. This means that the portion of storage addressable from one explicitly-named base register is often smaller than that addressable under implicit-base addressing. The other constraint is the key requirement that any register used as a base contain a suitable value at execution time. Clearly, the assembler cannot control what instruction will be executed to provide this value. The programmer, however, can write an instruction to load a base register. He must inform the assembler of what value will be in the base register at execution time, to permit it to generate addresses relative to the base register content. If there are multi-purpose registers which can serve either as a base or in another capacity, the programmer must indicate which registers are to be used as base registers.

For the IBM 360-370, the assembler instruction USING specifies a register as being available for use as a base, and states its execution-time content. The assembler instruction DROP withdraws the register from the list of those available for use as bases. Thus USING *,12 states that register 12 can be used as a base register and promises that its content will be the address of the current line. (That assembler instruction would typically be preceded by the machine instruction BALR 12,0 to load the ap-

propriate number at execution time into register 12, thus fulfilling the promise.)

If several registers have been made available for use as bases, the assembler, in addressing a given symbol, can use any base register for which the address displacement falls within the acceptable bounds. Consequently, the choice of base register is not fixed for the symbol, and is not among its attributes.

Symbol Attributes. One attribute we have seen is the address. This is the address of the start of the field to which the symbol refers. The address of each symbol is stored in the symbol table. Other attributes are normally stored there, too.

A particularly important attribute is whether a symbol is absolute or relative. This information is used by the loader in determining whether an adjustment is necessary when the load module is read into storage. It is used by the linker to determine the relocatability attribute of addresses which incorporate external references. Even within the assembler it is used to determine the relocatability attribute of address expressions.

If multiple location counters are permitted, the identity of the location counter to use for a symbol is another of its attributes. The location counter identity is used by the assembler to determine the origin with respect to which the symbol's relative address is to be generated.

The length attribute of a symbol is usually the length of the field named by the symbol. Different conventions for defining the length attribute are used in different assembler languages. Thus the storage reservation instruction ALPHA RSF 23 for a word-addressed machine might imply for the symbol ALPHA a length of 1 or a length of 23. The length attribute is used primarily in variable-field-length instructions to specify the length of the operands in the absence of an explicit length specification.

Some assembler languages permit references to symbol attributes. References to the length of a symbol may be explicit, as in L(ALPHA), or implicit in using ALPHA as an operand in a variable-field-length instruction. An attribute to which reference can often be made explicitly is the address of a symbol. This permits the programmer to specify a constant whose value will be the execution-time location associated with an assembler-language symbol. Such a constant is known as an *address constant* and is typically written A(FRONT) if FRONT is the symbol to whose location reference is made. The indirect addressing which results can be used in passing parameters, in implementing list structures, and in providing for base addressing.

Alternate Names. The provision of symbolic names for other fields of the instruction than storage addresses is often convenient and valuable. Register

designations, shift amounts, field lengths, and address displacements are usually encoded numerically in the assembler-language instruction. Program readability can often be enhanced by representing these numeric values symbolically. Thus a programmer might wish to use FR4 rather than 4 to name a floating-point register, and COMP, INCR, and LIMIT, rather than 8, 9, and 10, to name loop-control registers which hold comparand, increment, and limit. Assemblers which offer this *definitional* facility may use any of a variety of syntactic forms. Perhaps the most straightforward is the following use of the defined symbol as the label and the defining quantity as the operand.

```
FR4       SET     4
COMP      SET     8
INCR      SET     9
LIMIT     SET     10
```

Some assemblers offer a more general form of this facility, in which the operand may itself be a symbolic expression. Thus ZIP SET ZAP defines ZIP as a symbol whose attributes are set to equal the attributes which ZAP has at the point where the SET instruction is encountered. A subsequent change in the attributes of ZAP does not affect ZIP. This is analogous to call by value (see Section 3.5.2). Moreover, it is possible to redefine ZIP subsequently by use of another SET instruction. Except in the macro assemblers discussed in Section 5.4, the attributes of the defining expression will not change unless it incorporates a redefinable symbol. If the defining expression is just a numeric constant, the attribute is the value of the constant.

An apparently similar, but actually quite distinct, form of alternate naming caters to the dubious practice of two programmers independently preparing programs which are to be combined prior to assembly, yet using different names for the same entity. To avoid the tedious replacement of all occurrences of the name used in one of the programs, a *synonym* or name *equivalence* facility is provided in some assemblers. Thus ZIP EQU ZAP specifies that the symbols ZIP and ZAP are to be considered equivalent symbols throughout the program. Their attributes are the same. Thus whenever ZIP is used, the effect is the same as if ZAP had been named instead. This is analogous to call by name (see Section 3.5.3). It should be noted that in at least one system the mnemonic code EQU is used not for name equivalence, but for the definitional facility which we have called SET.

Alternate symbolic names for operation codes can be provided by EQU, although the use of a distinct instruction for this purpose offers some

advantages in processing. Thus the programmer who prefers MULT to the standard MD for double-precision floating-point multiplication on the IBM 360-370 can code MULT OPSYN MD to make MULT synonymous with MD whenever it is used as the operation code.

Each of the foregoing alternate naming facilities defines a symbol. It may or may not be required that the definition precede any use of a symbol so defined. Whether that requirement is imposed is dictated by whether lack of the definition will prevent the assembler from properly advancing the location counter during the first pass. The counter is not advanced, of course, when the SET, EQU, or OPSYN itself is encountered. Less restrictive alternate naming facilities can, of course, be provided at the cost of requiring more than two passes.

Literals. Often a programmer may wish to specify not the address of an operand but rather its value. In an instruction to increment a counter by 2, for example, the programmer may prefer just to write the constant "2" without concern for where that operand is located. An operand specified in this manner, with its value stated literally, is called a *literal*. Although several different instructions might include the same literal, usually only one copy of each literally specified value needs to be generated. The literals are placed together in a *pool* to avoid inserting each literal constant between instructions.

The literal pool is typically generated at the end of the object code. In some assembler languages, however, the programmer can control the placement of the pool. An instruction, perhaps LITORG, specifies that the pool of literals appearing up to that point in the text is to be placed at the address specified, usually the current value of the location counter. For machines with explicit-base addressing (e.g. IBM 360-370), this may be needed to ensure the addressability of literals. A literal pool generated subsequently includes only those literals which have been specified since the previous pool generation.

The assembler language must provide a distinction between numeric literals and numeric addresses, and between character literals and symbolic addresses. One way to distinguish is to use a different instruction format; another is to use a special symbol to mark a literal. Some literals, such as those with a decimal point or an exponent designator, are self-marking and would not need the special symbol. Even so, use of an explicit distinction permits a uniform implementation of all literals.

Error Checking. An assembler usually checks the source program for several different types of errors. One of the most important is the undefined symbol, which appears as an operand but nowhere as a label. Such an occurrence can be either rejected as an error or assumed to be an external

symbol. If the assembler language requires external symbols to be identified, then undefined symbols can be caught at assembly time. Otherwise, they are not caught until linkage is attempted.

A symbol which occurs as a label more than once has a different location counter value associated with each occurrence. Even if its other attributes are the same for each occurrence, its address is not. The resulting error is a multiply-defined symbol.

The detection of undefined and of multiply-defined symbols makes use of inter-statement context, as recorded in the symbol table. Many other invalidities can be checked during examination of one instruction alone. Among these are nonexistent operation codes, wrong number of operands, inappropriate operands, and a variety of syntax errors. Some aspects of the checking process may be simpler if the source language is constrained to a fixed, rather than a free, format.

Listing. The object code output is written onto a machine-readable medium, such as punched cards, paper tape, or magnetic disk. The programmer may require a human-readable listing of both source and object code, preferably side by side. To assist in debugging, the symbol table and a *concordance* or *cross-reference* table are usually printed. The symbol table lists each symbol together with its address, and perhaps other attributes. The cross-reference table indicates for each symbol where in the source program it is defined (used as a label) and where it is accessed (used in an operand specification). The two tables are usually sorted in alphabetic order for ease of use. Often they are combined in a single table, as in Fig. 2.9, which shows a listing of the assembly performed in Section 2.2. Such a listing can be produced, of course, by a separate listing program which accepts the source code, object code, and symbol table as inputs. Alternatively, it can be produced by the assembler itself during its second pass.

Error messages are an important component of the listing. Some assemblers group the error messages at the foot of the source-language program. Many programmers prefer, however, to have any message which is associated with a single source instruction printed next to that instruction. Even if this is not done, the message must at least identify the erroneous instruction. Some error messages, however, do not apply to a specific instruction and should not be interleaved with the program text.

A facility for comments is often provided. One method is to use a word or character reserved for the purpose (e.g. an asterisk in card column 1 to make the entire card a comment). Another method of identifying comments is to use an assembler instruction (say, COMMENT or REMARK) which specifies that the ensuing text is to be skipped by the assembler. The

Line	Label	Operation	Operand	Operand	Address	Length	Machine code
1		COPY	ZERO	OLDER	00	3	13 33 35
2		COPY	ONE	OLD	03	3	13 34 36
3		READ	LIMIT		06	2	12 38
4		WRITE	OLD		08	2	08 36
5	FRONT	LOAD	OLDER		10	2	03 35
6		ADD	OLD		12	2	02 36
7		STORE	NEW		14	2	07 37
8		SUB	LIMIT		16	2	06 38
9		BRPOS	FINAL		18	2	01 30
10		WRITE	NEW		20	2	08 37
11		COPY	OLD	OLDER	22	3	13 36 35
12		COPY	NEW	OLD	25	3	13 37 36
13		BR	FRONT		28	2	00 10
14	FINAL	WRITE	LIMIT		30	2	08 38
15		STOP			32	1	11
16	ZERO	CONST	0		33	1	00
17	ONE	CONST	1		34	1	01
18	OLDER	SPACE			35	1	XX
19	OLD	SPACE			36	1	XX
20	NEW	SPACE			37	1	XX
21	LIMIT	SPACE			38	1	XX

Address	Symbol	Definition	References
30	FINAL	14	9
10	FRONT	5	13
38	LIMIT	21	3 8 14
37	NEW	20	7 10 12
36	OLD	19	2 4 6 11 12
35	OLDER	18	1 5 11
34	ONE	17	2
33	ZERO	16	1

Figure 2.9 Assembly Listing

amount to be skipped may be fixed (e.g. the rest of the line) or specified as an operand. It can even be determined by the presence of another assembler instruction which terminates skipping. Neither of these methods is convenient for placing comments adjacent to program text. Such a capability can be provided in a number of ways. One is to reserve a field on each line for comments. Another is to reserve a character to indicate that the remainder of the line is a comment. Yet another is to treat as commentary all characters beyond the rightmost required field.

Assembler instructions may also be provided for control of the listing format, especially if the format of the source-language program is fixed.

Other options may include the printing of optional items, or even the suppression of part or all of the listing itself.

Assembler Control. Because assembler instructions other than those which reserve storage or define constants do not cause the generation of object code, there is not a one-to-one correspondence between source and object text. Consequently even an assembler for a machine in which all data and instruction lengths are fixed and equal (e.g. one word) requires two position counters, one for lines of source text and the other for addresses in the object program.

Some assembler instructions, such as those for alternate names, must be obeyed during the first pass of the assembler. Others, such as those for listing control, must be obeyed during the second pass. Yet others, such as scope delimiters, affect both passes. Sometimes a function specified by an assembler instruction can be implemented on either pass. An example is the generation of constants.

Repetitive Assembly. Sometimes a sequence of source-language statements are nearly identical to each other. This occurs commonly in the construction of tables. Much of the tedium of writing the full source code can be relieved by the provision of an assembler instruction, say REPEAT or ECHO, which causes one or more subsequent source lines to be assembled repeatedly with minor variations. Thus

```
        REPEAT 2,(1,10)
ARG$    CONST  $
FCT$    SPACE
```

might cause the group of two instructions to be repeated with the substitution, on each successive repetition, of the next digit in the range (1,10) for each occurrence of '$' in the group. This repetitive assembly would have the same effect as assembling the 20 following instructions.

```
ARG1    CONST  1
FCT1    SPACE
ARG2    CONST  2
FCT2    SPACE
        ---
ARG10   CONST  10
FCT10   SPACE
```

This facility is but a special case of a much more powerful facility, to be discussed in Chapter 4 and in Section 5.4, for generating source code at assembly time.

2.3.2 *Implementations*

The many functions described in the foregoing sections can be implemented straightforwardly in a two-pass assembler. As in our rudimentary assembler, Pass 1 is still concerned chiefly with symbol table construction and location counter management. Pass 2 still produces the object code and, if one is to be provided, the listing. The complexity of each pass depends, of course, on just which functions are selected. The basic actions for symbol table construction are listed in Fig. 2.10.

A major exception to the rules of that figure occurs if a symbol encountered as a label has no attributes to be entered into the table. How can this occur? If the associated operation is SET (define) or EQU (synonym). For the SET operation, the defining expression is evaluated and its attributes (e.g. address and relocatability mode) are entered into the table. This evaluation is similar to the evaluation of an address expression, which is described in the discussion of Pass 2. It is customary in a two-pass assembler to restrict the defining expression to contain no symbols which have themselves not yet been defined. Because of this restriction, the SET operation is most often used only to give symbolic names to constants.

If the label with no attributes is encountered in a synonym instruction EQU, it is possible to determine its attributes if the defining symbol has itself been previously defined. Because there is no guarantee, however, that referents of synonyms are defined in advance, a different approach is used uniformly for all occurrences of EQU. The label is entered into the symbol table and marked as being of type "EQU". The defining symbol is also placed in the same symbol table entry, perhaps in the address field if there is room. At the conclusion of Pass 1, the symbol table is scanned, and the defining symbol in each EQU type entry is replaced by a pointer to the table entry for the defining symbol. Thus all symbols defined as each other's synonyms are linked into a chain.

Whether or not a synonym function is provided, the symbol table can be scanned after the conclusion of Pass 1 for symbols without attributes.

Where encountered	Already in table?	Attributes in table?	Action taken
label	no		enter symbol and attributes
label	yes	no	enter attributes
label	yes	yes	*ERROR*: duplicate definition
operand	no		enter symbol
operand	yes		none

Figure 2.10 Rules for Constructing Symbol Table

These are flagged either as undefined symbols or as external symbols, accordingly as the explicit identification of external symbols is required or not. An alternative to the table scan after Pass 1 is to wait until Pass 2, when the absence of definition is obvious.

Location counter management is much simpler than symbol table construction. For a START operation the location counter is set either to the default (usually zero) or to a specified address; for an ORIGIN operation, to the specified address. For SPACE and CONST operations, the appropriate length is computed (if indeed any computation is necessary) and added to the location counter value in preparation for the next line. For other assembler instructions the location counter value remains unchanged. For a machine instruction, the instruction length is added to the location counter value. If instructions are of variable length, the operation code table will need to be consulted. For machines with storage alignment restrictions (e.g. IBM 360-370), an amount may need to be added to the location counter *before* associating its value with the current source line. If multiple location counters are provided, the foregoing description should be interpreted as referring to the location counter in use. The assembler instruction to change location counters merely selects the counter to be incremented subsequently.

Although location counter management and symbol table construction are the chief responsibilities of Pass 1, certain other functions are mandatory during that pass, and still others may optionally be performed during Pass 1 rather than Pass 2. A mandatory function which was not explicitly described in Section 2.3.1 is nevertheless implicit in any translation. This is the scanning of the source-language text to determine what it says. Both lexical and syntactic analysis are involved; they are discussed, not only for assemblers but for other translators as well, in Chapter 6.

Another mandatory function of Pass 1 is to examine operation fields sufficiently to determine the instruction length, which must be added to the location counter. If an operation code synonym facility (OPSYN) is provided, each OPSYN must be processed during the first pass to make the length of the defining instruction available. The OPSYN processing requires that an operation table entry be made either with a pointer to the defining operation code or with a duplicate of the table entry for the defining operation code. The remainder of operation code processing is optional during Pass 1, but is often included either a) if operation table lookup is performed anyway, or b) to effect the space saving of replacing the mnemonic operation code by the machine operation code. During Pass 1, then, the operation code may or may not be translated; determination of its format (if multiple formats exist) may or may not be performed.

Errors detected during Pass 1 must be recorded for incorporation in the listing. The generation of constants does not require a completed symbol table, and can be performed during either pass. The space required by the constants, however, does need to be determined during Pass 1 because of its effect on the location counter. Constant fields in an instruction often require little or no data conversion and may well be generated during Pass 1 if the instruction format has been determined.

Although it is possible to defer all processing of literals until Pass 2, unless LITORG is provided, it is usually more convenient to build the literal pool during Pass 1. Each time a literal is encountered, it is entered in a literal table, unless it is a duplicate of a literal already entered. Suppose that the character "@" marks a literal and that the following literals occur in the program in the order shown: @−1000, @1, @'TABLE', @12.75, @1, @3. Then the literal table might be organized as in Fig. 2.11, where lengths are given in bytes. Of course, it is not mandatory to eliminate duplicate literals, and some assemblers do generate a value for each occurrence of a given literal. The first literal of the pool can be assigned the address given by the value of the location counter at the conclusion of the first pass. Subsequent literals are assigned addresses determined by the lengths of the preceding literals.

Pass 1 typically transforms the source program into an *intermediate text* for input to Pass 2. Labels can be omitted, because code generation makes no use of them. Other symbols can be replaced, if desired, by pointers to the symbol table. The replacement is performed as each symbol is encountered, and is possible only if the symbol table construction algorithm does not change the position of a symbol once it has been entered. Although the use of pointers may result in some space savings, its chief advantage is elimination of table searching during Pass 2. In a similar manner, literals can be replaced by pointers to the literal table. Of course, if a source listing is to be produced, these labels, symbols, and literals cannot be discarded.

Although the symbols are no longer needed during Pass 2 for generating

Length	Value
4	-1000
4	1
5	'TABLE'
8	12.75
4	3

Figure 2.11 Example of Literal Table

code, they do serve another purpose, production of the concordance. This lists each symbol, the number of the source line on which it is defined, and the number of each line on which it is referenced. The attributes of each symbol are usually included also. For maximum usefulness, the concordance should be in alphabetic order. The required sorting of the symbol table can be performed at any time after the conclusion of Pass 1. A convenient time for an assembler which uses magnetic or paper tape is during tape rewind. If multitasking facilities are provided, sorting a copy of the symbol table can be assigned to a separate task to be executed concurrently with Pass 2. If symbols have been replaced by pointers in the intermediate text, sorting the symbol table will invalidate the pointers unless special action is taken. One solution is to keep an unsorted copy of the table for use in Pass 2. Another solution is based on use of the permutation vector generated by the sort process. An extra symbol table field stores, at a symbol's original position in the table, a pointer to its new position.

During Pass 2 the actual object code is generated. As the source (or intermediate) text is passed for a second time, the location counter is advanced just as it was during Pass 1. This time the purpose is not to assign addresses to symbols, but rather to incorporate addresses in the generated code. Operation codes are translated and instruction formats determined, to the extent that these functions were not carried out during Pass 1. Symbolic addresses which consist of a single symbol are easily handled. For a machine with direct or implicit-base addressing, the symbol is merely replaced by its location counter value from the symbol table. For a machine with explicit-base addressing, a base register table is used. Whenever a USING instruction is encountered during Pass 2, the number of the specified register and the promised value of its content are entered in the table. They remain until the content is changed by another USING or the register is deleted in response to a DROP. The symbol's location counter value *locn* is compared with the register content values *regc* listed in the base register table. There should be a register such that the displacement $disp = locn - regc$ falls in the appropriate range (0 to 4095 for the IBM 360-370). The value of *disp* and the number of the register are assembled into the object code. If there is more than one such register, then any one of them can be selected.

Addresses which correspond to redefinable local labels are easily generated during Pass 2 (or even Pass 1), provided that they are limited to backward references. For processing efficiency it is customary to use a fixed set of reserved symbols for this purpose. A fixed-size local label table holds the location counter value most recently associated with each redefin-

able symbol. Table lookup, whether for symbol redefinition or for address generation, is performed directly. Because no hashing or scan is required, this gains speed over use of the regular symbol table. Moreover, the existence of the small table of redefinable symbols permits substantial reduction in the size of the regular symbol table. This size reduction increases speed of access to regular symbols also.

Forward references to redefinable symbols cannot be resolved by the normal Pass 2 mechanism, because entries in the local label table are not fixed. The methods of Section 2.4 can be applied, however, as can the technique used in the UNISAP assembler for the Univac I. That assembler, apparently the first to provide redefinable local labels, used magnetic tape for the input, intermediate, and output texts. After the first pass, the intermediate text had to be repositioned for what we have described as Pass 2 processing. Instead of rewinding, the assembler made an extra pass, reading the intermediate tape backward. During this backward pass, what were originally forward references could be processed in the same manner as were backward references on a forward pass.

Address expressions are evaluated after lookup has determined the location counter value associated with each symbol. Thus LIMIT+3−FRONT in the context of the sample program of Section 2.2 would be evaluated as 31, and that value would be used in the instruction address field. A check must also be made that the address expression is not malformed with respect to relocatability; Chapter 8 explores that matter further.

Each reference to a literal is replaced by the address assigned to the corresponding entry in the literal table. The address correspondences can be established either between passes, or as the literals are encountered during Pass 2.

Other instruction fields, such as shift amounts, index register designations, and lengths, are usually stated explicitly as constants or as symbolic names which have been defined to represent constants. These are easily encoded. Sometimes an implicit specification (e.g. length in IBM 360-370) requires an attribute to be obtained from the symbol table.

The generation of data is another important responsibility of Pass 2. There is almost always a conversion from source-language encoding to machine encoding. The input form is generally a character string, composed perhaps of decimal digits, the period, plus and minus signs, and perhaps the letter E. The output form may be the machine encoding of a long-precision floating-point number, a short-precision integer, or a string of bits or of characters. For a rich assembler language, the description of data may well be expressed in what really amounts to a sublanguage of considerable size. The volume of assembler program code which performs the conversions may be very substantial indeed.

Whenever a CONST (or DC) instruction is encountered, the specified constant is generated at the attained point in the object code. For a SPACE (or DS) instruction, the assembler need only emit a directive to the loader to advance the location. As an alternative, the assembler can generate the required amount of fill, usually binary zeros. After the END instruction has been reached, the entries in the literal table are converted to machine encoding and appended. The alternative is to convert each literal as it is encountered, place the machine-representation literals in the literal table, and append the completed literal table to the end of the object code.

In generating the listing, the assembler needs access to the original text. It is possible, of course, to list the source text during Pass 1 and the object code during Pass 2, but the correspondences between the two representations then become difficult to establish. The listing normally includes source text image with line or sequence numbers, object code with location counter values, error messages, and a concordance. Assembler instructions which control printing may or may not be omitted from the listing, depending on the assembler. Whether to print them may itself be an option controlled by an assembler instruction! Much simpler for the assembler, of course, is to pass the needed files to a separate lister.

The object code produced by the assembler is still often called an object "deck", and may be said to be "punched" even on a system which uses backing storage rather than cards to hold the assembler output. The object code contains basically three kinds of information: machine-language code for the computer, address and relocation information for the loader, and a global symbol table for the linker. This last item may be a single table which includes both the symbols used within the module but not defined therein, and the symbols defined within the module and marked as being global. Alternatively, a separate table may be produced for each.

The organization of the assembler program must provide for a number of data structures. Among them are the following.

Pass 1 program
Pass 2 program
Source text
Intermediate text
Object text
Listing (if produced)
Symbol table
Global symbol table
Redefinable symbol table (if the function is provided)
Literal table
Machine instruction table

Assembler instruction table
Base register table (if explicit base registers are used)

The major differences among translation program organizations are determined by the choice of data structures to keep in main storage. The names of these choices customarily use the word "core" to refer to main storage, because of the widespread use of coincident-current magnetic cores to implement main storage. For many modern machines the term is technically not correct, but its brevity is appealing.

The *text-in-core* organization keeps the texts and tables in main storage and the programs in backing storage. Segments of the programs are brought in as needed. This organization is not very suitable for a two-pass program structure, but is often used for compilation programs, which may be composed of dozens of *phases.*

The *translator-in-core* organization keeps the Pass 1 and Pass 2 programs in main storage, along with the tables. The texts reside in backing storage. This organization is more suitable for an assembler, because the texts need to be accessed only serially, whereas such is not true of the program.

The *all-in-core* organization is just what its name implies. It is applicable, however, only if there is room for everything. This organization is typical of load-and-go assemblers, which are discussed in Section 2.4.1. It can be used, given enough space, for a two-pass assembler.

A typical organization for a two-pass assembler combines the translator-in-core approach with the provision of space for only the currently active pass. The texts are maintained in backing storage. The program starts with the Pass 1 program and tables in main storage and, after the termination of Pass 1, overwrites the Pass 1 program with the Pass 2 program, which has been waiting meanwhile in backing storage. The total main storage requirement is minimized by making the programs for the two passes approximately equal in size. This is accomplished by suitable allocation of those functions which can be performed on either pass.

2.4 ONE-PASS ASSEMBLERS

The translation performed by an assembler is essentially a collection of substitutions: machine operation code for mnemonic, machine address for symbolic, machine encoding of a number for its character representation, etc. Except for one factor, these substitutions could all be performed in one sequential pass over the source text. That factor is the forward ref-

erence. The separate passes of the two-pass assembler are required to handle forward references without restriction. If certain limitations are imposed, however, it becomes possible to handle forward references without making two passes. Different sets of restrictions lead to the one-pass module assembler and to the load-and-go assembler. These one-pass assemblers are particularly attractive when secondary storage is either slow or missing entirely, as on many small machines. The manual handling of punched cards or paper tape between passes is eliminated by eliminating a pass.

2.4.1 *Load-and-Go Assembler*

The *load-and-go* assembler forgoes the production of object code to generate absolute machine code and load it into the physical main storage locations from which it will be executed immediately upon completion of the assembly. The following restrictions are imposed by this mode of translation. (1) The program must run in a set of locations fixed at translation time; there is no relocation. (2) The program cannot be combined with one which has been translated separately. (3) Sufficient space is required in main storage to hold both the assembler and the machine-language program. Load-and-go assemblers are rather attractive for most student jobs, which are typically small and subject to frequent change.

Because the assembled program is in main storage, the assembler can fill in each forward reference as its definition is encountered. To do this, it is necessary to record the references to each undefined symbol. Because the number of such references is unpredictable, it is most convenient to organize the information as a chain. Each element of the chain includes a sign (to indicate whether the undefined symbol appears positively or negatively), the main storage location of the corresponding address field in the assembled program, and a pointer to the succeeding element in the chain for the same symbol. The last element includes a null pointer; a pointer to the first element (the head of the chain) occupies the symbol table entry for the undefined symbol.

The first occurrence of an undefined symbol causes the symbol to be entered in the symbol table, marked as undefined, with a pointer to a one-element chain. Each successive occurrence causes a new element to be inserted between the symbol table entry and the head of the previously created chain. The operand field which contains an undefined symbol is replaced in the assembled program by the value of those parts of the address expression other than undefined symbols. That value is zero, of course, if the operand consists of the undefined symbol alone. Figure 2.12

shows a portion of an assembler language program which makes forward references. Location counter values have been supplied. They are labeled "location" rather than "address" because for a load-and-go assembler they really do refer to physical storage locations.

The generated code (based on Fig. 2.3), symbol table, and undefined symbol chains are shown in Fig. 2.13 as they stand after the STORE instruction has been translated. In the illustration, chain elements are assumed to require two words each, with space starting at location 120 available for the chain. Other implementations of the chain are also possible.

Location	Label	Operation	Operand 1
12		READ	PV
--		---	
47		LOAD	PV
49		ADD	THERM+1
51		STORE	PV
--		---	
92	PV	SPACE	
93	THERM	CONST	386.2
94		CONST	374.9

Figure 2.12 Input to Load-and-Go Assembler (location added)

Location	Machine code
12	12 00
--	---
47	03 00
49	02 01
51	07 00

(a) Assembled program

Symbol	Marker	Address
PV	undefined	126
THERM	undefined	124

(b) Symbol table

Location	Sign	Address	Pointer
120	+	13	null
122	+	48	120
124	+	50	null
126	+	52	122

(c) Undefined symbol chains

Figure 2.13 Data Structures after Translation of Location 51

When the definition of a previously referenced symbol is finally encountered, its storage location is added to or subtracted from the machine code which has already been generated at the location specified in each element of the associated chain, as directed by the element's sign field. When the end of the chain is reached, the mark in the symbol table entry is reset from "undefined" to "defined", and the pointer to the first chain element is replaced by the symbol's absolute location, which is the now-known location counter value. The space occupied by the undefined symbol chain is returned to the available space list for further use. The generated code and symbol table are shown in Fig. 2.14 as they stand after the symbol THERM has been processed. The entry "XX" for the machine code generated at location 92 indicates that the content of that location is unspecified.

A particularly economical implementation of the chain of undefined symbols stores the pointers within the partially translated program itself. The operand field which contains an undefined symbol is replaced in the assembled program by a pointer to the previous use of that symbol. Because this precludes making any provision for other parts of an address expression, a further restriction is necessary. Undefined symbols may then not appear in address expressions. Thus BR REPLACE+6 is permitted only if REPLACE occurs as a label earlier in the source program.

Although there is, to be sure, no full Pass 2, the load-and-go assembler is not really a pure one-pass assembler. The chain-following actions really do constitute partial second passes. Because the portions of the program which they examine are necessarily still in main storage, the cost often associated with a conventional second pass is nevertheless avoided.

Location	Machine code
12	12 92
--	---
47	03 92
49	02 94
51	07 92
--	---
92	XX
93	386.2

(a) Assembled program

Symbol	Marker	Address
PV	defined	92
THERM	defined	93

(b) Symbol table

Figure 2.14 Data Structures after Translation of Location 93

2.4.2 *One-Pass Module Assembler*

A *module* assembler, unlike a load-and-go assembler, produces not machine code, but rather object code which can later be linked and loaded. The one-pass module assembler purports to accomplish this in a single pass, despite forward references. The strategy is to rely on the pass which will eventually be made over the program by the loader, and to use that as the second assembler pass for those functions which just cannot be performed in one pass. Thus more work is imposed on the loader, but the assembler requires only one pass. The restriction typically imposed on source programs is the prohibition of forward references other than branches. Thus data areas precede the instructions which reference them, and literals cannot be used.

Each undefined symbol must occur in a branch address. It is entered in a *branch-ahead table* along with its sign and the address of the instruction address field in which it appears. If several branches are expected to refer to the same undefined symbol, the branch-ahead table could be implemented as a collection of chains similar to the undefined symbol chains described in Section 2.4.1. Because the number of branches to undefined symbols is usually not great, however, it is probably simpler to omit the pointers and just use a conventional table. Each time a label is encountered, the symbol and its attributes are entered into the symbol table. The branch-ahead table is then scanned for all occurrences of that symbol. For each occurrence, the assembler first generates a directive to the loader to adjust the corresponding address field when the program is loaded, and then deletes the entry from the branch-ahead table.

FOR FURTHER STUDY

Barron [1972] is an excellent short book on simple and macro assemblers, loaders, and linkers. Its Chapters 2, 4, and 6 are devoted particularly to assemblers. Two good brief treatments of assemblers are Chapter 4 of Ullman [1976] and Section 8.3 of Brooks and Iverson [1969], which is insightful but specialized to the IBM 360. More extensive presentations are in Chapter 9 of Gear [1974] and Chapter 4 of Graham [1975].

EXERCISES

2.1 What program development costs rise faster than proportionally to the length of a program component?

2.2 Rewrite the program of Fig. 2.2 in a form which makes no forward references to either instructions or data.

2.3 Assemble the following program manually, showing the resulting object code and the symbol table. Use starting location 100.

```
            READ    N
            COPY    ONE       FACT
            COPY    ONE       IDX
HEAD        LOAD    N
            SUB     IDX
            BRZERO  ALL
            BRNEG   ALL
            LOAD    IDX
            ADD     ONE
            STORE   IDX
            MULT    FACT
            STORE   FACT
            BR      HEAD
ALL         WRITE   FACT
            STOP
N           SPACE
IDX         SPACE
FACT        SPACE
ONE         CONST   1
```

2.4 Disassemble the following object code manually, showing a possible assembler-language representation and the symbol table. Be a careful detective in analyzing the last line.

```
00      3       13 22 23
03      2       12 24
05      2       03 24
07      2       05 21
09      2       06 23
11      2       01 15
13      2       00 03
15      2       03 24
17      2       07 23
19      2       00 03
21      1       11
22      1       00
23      2       08 03
```

2.5 Let @ mark a literal and A(symbol) be an address constant. Using Fig. 2.3, show the generated code which corresponds to the following source text.

```
            LOAD    @A(FRONT)
            ADD     @3
            STORE   NEXT
FRONT       CONST   99
NEXT        SPACE
```

2.6 Explain why the four occurrences of "XX" cannot be omitted from the end of the machine code in the last paragraph of Section 2.2.2.

2.7 Write an address expression to designate the start of the Jth full word of the six full words in the area reserved by the IBM 360 Assembler instruction WORK DS 15D,6F (see "Storage Reservation" in Section 2.3.1).

2.8 Let @ mark a literal and A(symbol) be an address constant. Using Fig. 2.3, show the generated object code which corresponds to the following source text. (The program is not intended to be useful.)

```
          READ    OFFSET
          LOAD    OFFSET
          ADD     @A(FAR)
          STORE   NEAR
          BR      FAR+1
NEAR      SPACE
OFFSET    SPACE
FAR       CONST   A(NEAR)
```

2.9 Let EQU be a synonym facility, SET a definition facility, and * the current value of the location counter. Explain (a) the difference between the following instructions, and (b) how each can be implemented in a two-pass assembler.

```
BACK6     EQU     *-6
BACK6     SET     *-6
```

2.10 Consider the chain of symbols which are defined by a synonym facility to be equivalent. Why is the chain usually not constructed during Pass 1 as the symbols are read?

2.11 Let OPSYN be the operation code synonym facility mentioned in the third paragraph under "Alternate Names" in Section 2.3.1.

a) How is it distinct from EQU or SET?
b) How can it be implemented?
c) Why is it advantageous to require *all* occurrences of OPSYN to precede *any* instructions?

2.12 What are the advantages and disadvantages of holding symbolic operation codes in the main symbol table?

2.13 Let LITORG be an instruction to assign the current location counter value as the origin of a literal pool. Design an implementation of literals, including LITORG, and state precisely and fully what actions are performed during each pass. Make sure that your implementation is capable of handling the following situation (where @ marks a literal).

```
                ---     @A(PLACE)
                ---
                LITORG
                ---
PLACE           ---
```

2.14 Design an implementation of literals (without LITORG) which does all the processing on Pass 2.

2.15 [Donovan] Explain how to process LITORG on Pass 1 only, given that address constants are not permitted.

2.16 Suppose that a program contains both the literal @3 and an instruction C3 CONST 3. Is it permissible to assign the same location to the literal as to C3?

2.17 How can an assembler be designed to permit the use in a literal of a symbol defined by the SET instruction to represent a given value, as in the following?

```
LIMIT   SET     4
        ---
        ADD     @LIMIT
        ---
LIMIT   SET     6
        ---
        DIVIDE  @LIMIT
```

2.18 If the intermediate text contains pointers to the symbol table, can the symbols themselves be dropped from the table before Pass 2?

2.19 [Donovan] To permit a two-pass assembler to generate code for absolute loading (i.e. without relocation), the assembler instruction ABS has been defined. Its one operand specifies the execution-time physical location of the origin of the module being assembled. Where may the ABS instruction appear in the source program? On which pass(es) would the ABS be processed and how?

2.20 How can literals be processed by a load-and-go assembler?

2.21 Consider the translation of the program of Fig. 2.2 by a load-and-go assembler. Assume starting location 0. Show the data structures, as in Figs. 2.12-2.14,

a) after the translation of line 5;
b) after the translation of line 14;
c) after the translation of line 21.

2.22 Consider the translation of the program of Fig. 2.2 by a load-and-go assembler which uses the implementation described in the penultimate paragraph of Section 2.4.1. Assuming starting location 100, show the symbol table and the assembled machine-language program

a) after the translation of line 5;
b) after the translation of line 18.

2.23 Let the operand in location 50 of Fig. 2.12 be changed to THERM. Illustrate the implementation described in the penultimate paragraph of Section 2.4.1 by showing the assembled program and symbol table
a) after the translation of location 50;
b) after the translation of location 92.

2.24 Why must a one-pass module assembler forgo literals?

2.25 How can a one-pass module assembler handle branches to external symbols?