7

THE ASSEMBLY LANGUAGE LEVEL

In Chapters 4, 5, and 6 we discussed three different levels present on most contemporary computers. This chapter is concerned primarily with another level that is present on all computers: the assembly language level. The assembly language level differs in a significant respect from the microarchitecture, ISA, and operating system machine levels—it is implemented by translation rather than by interpretation.

Programs that convert a user's program written in some language to another language are called **translators**. The language in which the original program is written is called the **source language** and the language to which it is converted is called the **target language**. Both the source language and the target language define levels. If a processor that can directly execute programs written in the source language is available, there is no need to translate the source program into the target language.

Translation is used when a processor (either hardware or an interpreter) is available for the target language but not for the source language. If the translation has been performed correctly, running the translated program will give precisely the same results as the execution of the source program would have given had a processor for it been available. Consequently, it is possible to implement a new level for which there is no processor by first translating programs written for that level to a target level and then executing the resulting target-level programs.

It is important to note the difference between translation, on the one hand, and interpretation, on the other hand. In translation, the original program in the source

language is not directly executed. Instead, it is converted to an equivalent program called an **object program** or **executable binary program** whose execution is carried out only after the translation has been completed. In translation, there are two distinct steps:

1. Generation of an equivalent program in the target language.
2. Execution of the newly generated program.

These two steps do not occur simultaneously. The second step does not begin until the first has been completed. In interpretation, there is only one step: executing the original source program. No equivalent program need be generated first, although sometimes the source program is converted to an intermediate form (e.g., Java byte code) for easier interpretation.

While the object program is being executed, only three levels are in evidence: the microarchitecture level, the ISA level, and the operating system machine level. Consequently, three programs—the user's object program, the operating system, and the microprogram (if any)—can be found in the computer's memory at run time. All traces of the original source program have vanished. Thus the number of levels present at execution time may differ from the number of levels present before translation. It should be noted, however, that although we define a level by the instructions and linguistic constructs available to its programmers (and not by the implementation technique), other authors sometimes make a greater distinction between levels implemented by execution-time interpreters and levels implemented by translation.

7.1 INTRODUCTION TO ASSEMBLY LANGUAGE

Translators can be roughly divided into two groups, depending on the relationship between the source language and the target language. When the source language is essentially a symbolic representation for a numerical machine language, the translator is called an **assembler** and the source language is called an **assembly language**. When the source language is a high-level language such as Java or C and the target language is either a numerical machine language or a symbolic representation for one, the translator is called a **compiler**.

7.1.1 What Is an Assembly Language?

A pure assembly language is a language in which each statement produces exactly one machine instruction. In other words, there is a one-to-one correspondence between machine instructions and statements in the assembly program. If each line in the assembly language program contains exactly one statement and

each machine word contains exactly one machine instruction, then an *n*-line assembly program will produce an *n*-instruction machine language program.

The reason that people use assembly language, as opposed to programming in machine language (in binary or hexadecimal), is that it is much easier to program in assembly language. The use of symbolic names and symbolic addresses instead of binary or hexadecimal ones makes an enormous difference. Most people can remember that the abbreviations for add, subtract, multiply, and divide are ADD, SUB, MUL, and DIV, but few can remember the corresponding numerical values the machine uses. The assembly language programmer need only remember the symbolic names because the assembler translates them to the machine instructions.

The same remarks apply to addresses. The assembly language programmer can give symbolic names to memory locations and have the assembler worry about supplying the correct numerical values. The machine language programmer must always work with the numerical values of the addresses. As a consequence, no one programs in machine language today, although people did so decades ago, before assemblers had been invented.

Assembly languages have another property, besides the one-to-one mapping of assembly language statements onto machine instructions, that distinguishes them from high-level languages. The assembly programmer has access to all the features and instructions available on the target machine. The high-level language programmer does not. For example, if the target machine has an overflow bit, an assembly language program can test it, but a Java program cannot test it. An assembly language program can execute every instruction in the instruction set of the target machine, but the high-level language program cannot. In short, everything that can be done in machine language can be done in assembly language, but many instructions, registers, and similar features are not available for the high-level language programmer to use. Languages for system programming, like C, are a cross between these types, with the syntax of a high-level language but with some of the access to the machine of an assembly language.

One final difference that is worth making explicit is that an assembly language program can run only on one family of machines, whereas a program written in a high-level language can potentially run on many machines. For many applications, this ability to move software from one machine to another is of great practical importance.

7.1.2 Why Use Assembly Language?

Assembly language programming is difficult. Make no mistake about that. It is not for wimps and weaklings. Furthermore, writing a program in assembly language takes much longer than writing the same program in a high-level language. It also takes much longer to debug and is much harder to maintain.

Under these conditions, why would anyone ever program in assembly language? There are two reasons: performance and access to the machine. First of

all, an expert assembly language programmer working very hard can sometimes produce code that is much smaller and much faster than a high-level language programmer can. For some applications, speed and size are critical. Many embedded applications, such as the code on a smart card or RFID card, device drivers, string-manipulation libraries, BIOS routines, and the inner loops of performance-critical real-time applications fall in this category.

Second, some procedures need complete access to the hardware, something usually impossible in high-level languages. For example, the low-level interrupt and trap handlers in an operating system and the device controllers in many embedded real-time systems fall into this category.

Besides these reasons for programming in assembly language, there are also two additional reasons for studying it. First, a compiler must either produce output used by an assembler or perform the assembly process itself. Thus understanding assembly language is essential to understanding how compilers work. Someone has to write the compiler (and its assembler) after all.

Second, studying assembly language exposes the real machine to view. For students of computer architecture, writing some assembly code is the only way to get a feel for what a machine is really like at the architectural level.

7.1.3 Format of an Assembly Language Statement

Although the structure of an assembly language statement closely mirrors the structure of the machine instruction that it represents, assembly languages for different machines sufficiently resemble one another to allow a discussion of assembly language in general. Figure 7-1 shows fragments of assembly language programs for the x86 which performs the computation $N = I + J$. The statements below the blank line are commands to the assembler to reserve memory for the variables I, J, and N and are not symbolic representations of machine instructions.

Label	Opcode	Operands	Comments
FORMULA:	MOV	EAX,I	; register EAX = I
	ADD	EAX,J	; register EAX = I + J
	MOV	N,EAX	; N = I + J
I	DD	3	; reserve 4 bytes initialized to 3
J	DD	4	; reserve 4 bytes initialized to 4
N	DD	0	; reserve 4 bytes initialized to 0

Figure 7-1. Computation of $N = I + J$ on the x86.

Several assemblers exist for the Intel family (i.e., x86), each with a different syntax. In this chapter we will use the Microsoft MASM assembly language for our example. There are many assemblers for the ARM, but the syntax is comparable to the x86 assembler, so one example should siffice.

Assembly language statements have up to four parts: first, a label field, second, an operation (opcode) field, third, an operand field, and fourth, a comments field. None of these is mandatory. Labels, which are used to provide symbolic names for memory addresses, are needed on executable statements so that the statements can be branched to. Additionally, they are needed for data words to permit the data stored there to be accessible by symbolic name. If a statement is labeled, the label (usually) begins in column 1.

The example of Fig. 7-1 has four labels: *FORMULA*, *I*, *J*, and *N*. The MASM requires colons on code labels but not on data labels. There is nothing fundamental about this. Other assemblers may have other requirements. Nothing in the underlying architecture suggests one choice or the other. One advantage of the colon notation is that with it a label can appear by itself on a line, with the opcode in column 1 of the next line. This style is convenient for compilers to generate. Without the colon, there would be no way to tell a label on a line all by itself from an opcode on a line all by itself. The colon eliminates this potential ambiguity.

It is an unfortunate characteristic of some assemblers that labels are restricted to six or eight characters. In contrast, most high-level languages allow the use of arbitrarily long names. Long, well-chosen names make programs much more readable and understandable by other people.

Each machine has some registers, so they need names. The x86 registers have names like EAX, EBX, ECX, and so on.

The opcode field contains either a symbolic abbreviation for the opcode—if the statement is a symbolic representation for a machine instruction—or a command to the assembler itself. The choice of an appropriate name is just a matter of taste, and different assembly language designers often make different choices. The designers of the MASM assembler decided to use MOV for both loading a register from memory and storing a register into memory but they could have chosen MOVE or LOAD and STORE.

Assembly progams often need to reserve space for variables. The MASM assembly language designers chose DD (Define Double), since a word on the 8088 was 16 bits.

The operand field of an assembly language statement is used to specify the addresses and registers used as operands by the machine instruction. The operand field of an integer addition instruction tells what is to be added to what. The operand field of a branch instruction tells where to branch to. Operands can be registers, constants, memory locations, and so on.

The comments field provides a place for programmers to put helpful explanations of how the program works for the benefit of other programmers who may subsequently use or modify the program (or for the benefit of the original programmer a year later). An assembly language program without such documentation is nearly incomprehensible to all programmers, frequently including the author as well. The comments field is solely for human consumption; it has no effect on the assembly process or on the generated program.

7.1.4 Pseudoinstructions

In addition to specifying which machine instructions to execute, an assembly language program can also contain commands to the assembler itself, for example, asking it to allocate some storage or to eject to a new page on the listing. Commands to the assembler itself are called **pseudoinstructions** or sometimes **assembler directives**. We have already seen a typical pseudoinstruction in Fig. 7-1(a): DD. Some other pseudoinstructions are listed in Fig. 7-2. These are taken from the Microsoft MASM assembler for the x86.

Pseudoinstruction	Meaning
SEGMENT	Start a new segment (text, data, etc.) with certain attributes
ENDS	End the current segment
ALIGN	Control the alignment of the next instruction or data
EQU	Define a new symbol equal to a given expression
DB	Allocate storage for one or more (initialized) bytes
DW	Allocate storage for one or more (initialized) 16-bit (word) data items
DD	Allocate storage for one or more (initialized) 32-bit (double) data items
DQ	Allocate storage for one or more (initialized) 64-bit (quad) data items
PROC	Start a procedure
ENDP	End a procedure
MACRO	Start a macro definition
ENDM	End a macro definition
PUBLIC	Export a name defined in this module
EXTERN	Import a name from another module
INCLUDE	Fetch and include another file
IF	Start conditional assembly based on a given expression
ELSE	Start conditional assembly if the IF condition above was false
ENDIF	End conditional assembly
COMMENT	Define a new start-of-comment character
PAGE	Generate a page break in the listing
END	Terminate the assembly program

Figure 7-2. Some of the pseudoinstructions available in the MASM assembler (MASM).

The SEGMENT pseudoinstruction starts a new segment, and ENDS terminates one. It is allowed to start a text segment, with code, then start a data segment, then go back to the text segment, and so on.

ALIGN forces the next line, usually data, to an address that is a multiple of its argument. For example, if the current segment has 61 bytes of data already, then after ALIGN 4 the next address allocated will be 64.

EQU is used to give a symbolic name to an expression. For example, after the pseudoinstruction

```
BASE  EQU  1000
```

the symbol BASE can be used everywhere instead of 1000. The expression that follows the EQU can involve multiple defined symbols combined with arithmetic and other operators, as in

```
LIMIT  EQU  4 * BASE + 2000
```

Most assemblers, including MASM, require that a symbol be defined before it is used in an expression like this.

The next four pseudoinstructions, DB, DW, DD, and DQ, allocate storage for one or more variables of size 1, 2, 4, or 8 bytes, respectively. For example,

```
TABLE DB 11, 23, 49
```

allocates space for 3 bytes and initializes them to 11, 23, and 49, respectively. It also defines the symbol *TABLE* and sets it equal to the address where 11 is stored.

The PROC and ENDP pseudoinstructions define the beginning and end of assembly language procedures, respectively. Procedures in assembly language have the same function as procedures in other programming languages. Similarly, MACRO and ENDM delimit the scope of a macro definition. We will study macros later in this chapter.

The next two pseudoinstructions, PUBLIC and EXTERN, control the visibility of symbols. It is common to write programs as a collection of files. Frequently, a procedure in one file needs to call a procedure or access a data word defined in another file. To make this cross-file referencing possible, a symbol that is to be made available to other files is exported using PUBLIC. Similarly, to prevent the assembler from complaining about the use of a symbol that is not defined in the current file, the symbol can be declared as EXTERN, which tells the assembler that it will be defined in some other file. Symbols that are not declared in either of these pseudoinstructions have a scope of the local file. This default means that using, say, *FOO* in multiple files does not generate a conflict because each definition is local to its own file.

The INCLUDE pseudoinstruction causes the assembler to fetch another file and include it bodily into the current one. Such included files often contain definitions, macros, and other items needed in multiple files.

Many assemblers, support conditional assembly. For example,

```
WORDSIZE EQU 32
IF WORDSIZE GT 32
WSIZE:    DD   64
ELSE
WSIZE:    DD   32
ENDIF
```

allocates a single 32-bit word and calls its address *WSIZE*. The word is initialized to either 64 or 32, depending on the value of *WORDSIZE*, in this case, 32. Typically this construction would be used to write a program that could be assembled for either 32-bit mode or 64-bit mode. IF and ENDIF, then by changing a single definition, *WORDSIZE*, the program can automatically be set to assemble for either size. Using this approach, it is possible to maintain one source program for multiple (different) target machines, which makes software development and maintenance easier. In many cases, all the machine-dependent definitions, like *WORDSIZE*, are collected into a single file, with different versions for different machines. By including the right definitions file, the program can be easily recompiled for different machines.

The COMMENT pseudoinstruction allows the user to change the comment delimiter to something other than semicolon. PAGE is used to control the listing the assembler can produce, if requested. END marks the end of the program.

Many other pseudoinstructions exist in MASM. Other x86 assemblers have a different collection of pseudoinstructions available because they are dictated not by the underlying architecture, but by the taste of the assembler writer.

7.2 MACROS

Assembly language programmers frequently need to repeat sequences of instructions several times within a program. The most obvious way to do so is simply to write the required instructions wherever they are needed. If a sequence is long, however, or must be used a large number of times, writing it repeatedly becomes tedious.

An alternative approach is to make the sequence into a procedure and call it wherever it is needed. This strategy has the disadvantage of requiring a procedure call instruction and a return instruction to be executed every time a sequence is needed. If the sequences are short—for example, two instructions—but are used frequently, the procedure call overhead may significantly slow the program down. Macros provide an easy and efficient solution to the problem of repeatedly needing the same or nearly the same sequences of instructions.

7.2.1 Macro Definition, Call, and Expansion

A **macro definition** is a way to give a name to a piece of text. After a macro has been defined, the programmer can write the macro name instead of the piece of program. A macro is, in effect, an abbreviation for a piece of text. Figure 7-3(a) shows an assembly language program for the x86 that exchanges the contents of the variables *p* and *q* twice. These sequences could be defined as macros, as shown in Fig. 7-3(b). After its definition, every occurrence of *SWAP* causes it to be replaced by the four lines:

```
MOV EAX,P
MOV EBX,Q
MOV Q,EAX
MOV P,EBX
```

The programmer has defined *SWAP* as an abbreviation for the four statements shown above.

```
MOV   EAX,P
MOV   EBX,Q
MOV   Q,EAX
MOV   P,EBX

MOV   EAX,P
MOV   EBX,Q
MOV   Q,EAX
MOV   P,EBX
```

(a)

```
SWAP  MACRO
      MOV EAX,P
      MOV EBX,Q
      MOV Q,EAX
      MOV P,EBX
      ENDM

      SWAP

      SWAP
```

(b)

Figure 7-3. Assembly language code for interchanging P and Q twice. (a) Without a macro. (b) With a macro.

Although different assemblers have slightly different notations for defining macros, all require the same basic parts in a macro definition:

1. A macro header giving the name of the macro being defined.
2. The text that is the body of the macro.
3. A pseudoinstruction marking the end of the definition (e.g., ENDM).

When the assembler encounters a macro definition, it saves it in a macro definition table for subsequent use. From that point on, whenever the name of the macro (*SWAP* in the example of Fig. 7-3) appears as an opcode, the assembler replaces it by the macro body. The use of a macro name as an opcode is known as a **macro call** and its replacement by the macro body is called **macro expansion**.

Macro expansion occurs during the assembly process and not during execution of the program. This point is important. The program of Fig. 7-3(a) and that of Fig. 7-3(b) will produce precisely the same machine language code. Looking only at the machine language program, it is impossible to tell whether or not any macros were involved in its generation. The reason is that once macro expansion has been completed the macro definitions are discarded by the assembler. No trace of them is left in the generated program.

Macro calls should not be confused with procedure calls. The basic difference is that a macro call is an instruction to the assembler to replace the macro name

with the macro body. A procedure call is a machine instruction that is inserted into the object program and that will later be executed to call the procedure. Figure 7-4 compares macro calls with procedure calls.

Item	Macro call	Procedure call
When is the call made?	During assembly	During program execution
Is the body inserted into the object program every place the call is made?	Yes	No
Is a procedure call instruction inserted into the object program and later executed?	No	Yes
Must a return instruction be used after the call is done?	No	Yes
How many copies of the body appear in the object program?	One per macro call	One

Figure 7-4. Comparison of macro calls with procedure calls.

Conceptually, it is best to think of the assembly process as taking place in two passes. On pass one, all the macro definitions are saved and the macro calls expanded. On pass two, the resulting text is processed as though it was in the original program. In this view, the source program is read in and is then transformed into another program from which all macro definitions have been removed, and in which all macro calls have been replaced by their bodies. The resulting output, an assembly language program containing no macros at all, is then fed into the assembler.

It is important to keep in mind that a program is a string of characters including letters, digits, spaces, punctuation marks, and "carriage returns" (change to a new line). Macro expansion consists of replacing certain substrings of this string with other character strings. A macro facility is a technique for manipulating character strings, without regard to their meaning.

7.2.2 Macros with Parameters

The macro facility previously described can be used to shorten source programs in which precisely the same sequence of instructions occurs repeatedly. Frequently, however, a program contains several sequences of instructions that are almost but not quite identical, as illustrated in Fig. 7-5(a). Here the first sequence exchanges *P* and *Q*, and the second sequence exchanges *R* and *S*.

Macro assemblers handle the case of nearly identical sequences by allowing macro definitions to provide **formal parameters** and by allowing macro calls to supply **actual parameters**. When a macro is expanded, each formal parameter

```
MOV     EAX,P
MOV     EBX,Q
MOV     Q,EAX
MOV     P,EBX

MOV     EAX,R
MOV     EBX,S
MOV     S,EAX
MOV     R,EBX
```

(a)

```
CHANGE  MACRO P1, P2
        MOV EAX,P1
        MOV EBX,P2
        MOV P2,EAX
        MOV P1,EBX
        ENDM

        CHANGE P, Q

        CHANGE R, S
```

(b)

Figure 7-5. Nearly identical sequences of statements. (a) Without a macro. (b) With a macro.

appearing in the macro body is replaced by the corresponding actual parameter. The actual parameters are placed in the operand field of the macro call. Figure 7-5(b) shows the program of Fig. 7-5(a) rewritten using a macro with two parameters. The symbols *P1* and *P2* are the formal parameters. Each occurrence of *P1* within a macro body is replaced by the first actual parameter when the macro is expanded. Similarly, *P2* is replaced by the second actual parameter. In the macro call

```
CHANGE P, Q
```

P is the first actual parameter and *Q* is the second actual parameter. Thus the executable programs produced by both parts of Fig. 7-5 are identical. They contain precisely the same instructions with the same operands.

7.2.3 Advanced Features

Most macro processors have a whole raft of advanced features to make life easier for the assembly language programmer. In this section we will take a look at a few of MASM's advanced features. One problem that occurs with all assemblers that support macros is label duplication. Suppose that a macro contains a conditional branch instruction and a label that is branched to. If the macro is called two or more times, the label will be duplicated, causing an assembly error. One solution is to have the programmer supply a different label on each call as a parameter. A different solution (used by MASM) is to allow a label to be declared LOCAL, with the assembler automatically generating a different label on each expansion of the macro. Some other assemblers have a rule that numeric labels are automatically local.

MASM and most other assemblers allow macros to be defined within other macros. This feature is most useful in combination with conditional assembly.

Typically, the same macro is defined in both parts of an IF statement, like this:

```
M1   MACRO
     IF WORDSIZE GT 16
M2        MACRO
          ...
          ENDM
     ELSE
M2        MACRO
     ...
          ENDM
     ENDIF
          ENDM
```

Either way, the macro *M2* will be defined, but the definition will depend on whether the program is being assembled on a 16-bit machine or a 32-bit machine. If *M1* is not called, *M2* will not be defined at all.

Finally, macros can call other macros, including themselves. If a macro is recursive, that is, it calls itself, it must pass itself a parameter that is changed on each expansion and the macro must test the parameter and terminate the recursion when it reaches a certain value. Otherwise the assembler can be put into an infinite loop. If this happens, the assembler must be killed explicitly by the user.

7.2.4 Implementation of a Macro Facility in an Assembler

To implement a macro facility, an assembler must be able to perform two functions: save macro definitions and expand macro calls. We will examine these now.

The assembler must maintain a table of all macro names and, along with each name, a pointer to its stored definition so that it can be retrieved when needed. Some assemblers have a separate table for macro names and some have a combined opcode table in which all machine instructions, pseudoinstructions, and macro names are kept.

When a macro definition is encountered, a table entry is made giving the name of the macro, the number of formal parameters, and a pointer to another table—the macro definition table—where the macro body will be kept. A list of the formal parameters is also constructed at this time for use in processing the definition. The macro body is then read and stored in the macro definition table. Formal parameters occurring within the body are indicated by some special symbol. For example, the internal representation of the macro definition of *CHANGE* with semicolon as "carriage return" and ampersand as the formal parameter symbol might be:

```
MOV EAX,&P1; MOV EBX,&P2; MOV &P2,EAX; MOV &P1,EBX;
```

Within the macro definition table the macro body is simply a character string.

During pass one of the assembly, opcodes are looked up and macros expanded. Whenever a macro definition is encountered, it is stored in the macro table. When a macro is called, the assembler temporarily stops reading input from the input device and starts reading from the stored macro body instead. Formal parameters extracted from the stored macro body are replaced by the actual parameters provided in the call. The presence of an ampersand in front of the formal parameters makes it easy for the assembler to recognize them.

7.3 THE ASSEMBLY PROCESS

In the following sections we will briefly describe how an assembler works. Although each machine has a different assembly language, the assembly process is sufficiently similar on different machines that it is possible to describe it in general.

7.3.1 Two-Pass Assemblers

Because an assembly language program consists of a series of one-line statements, it might at first seem natural to have an assembler that read one statement, then translated it to machine language, and finally output the generated machine language onto a file, along with the corresponding piece of the listing, if any, onto another file. This process would then be repeated until the whole program had been translated. Unfortunately, this strategy does not work.

Consider the situation where the first statement is a branch to *L*. The assembler cannot assemble this statement until it knows the address of statement *L*. Statement *L* may be near the end of the program, making it impossible for the assembler to find the address without first reading almost the entire program. This difficulty is called the **forward reference problem**, because a symbol, *L*, has been used before it has been defined; that is, a reference has been made to a symbol whose definition will only occur later.

Forward references can be handled in two ways. First, the assembler may in fact read the source program twice. Each reading of the source program is called a **pass**; any translator that reads the input program twice is called a **two-pass translator**. On pass one, the definitions of symbols, including statement labels, are collected and stored in a table. By the time the second pass begins, the values of all symbols are known; thus no forward reference remains and each statement can be read, assembled, and output. Although this approach requires an extra pass over the input, it is conceptually simple.

The second approach consists of reading the assembly program once, converting it to an intermediate form, and storing this intermediate form in a table in memory. Then a second pass is made over the table instead of over the source program. If there is enough memory (or virtual memory), this approach saves I/O time. If a listing is to be produced, then the entire source statement, including all

the comments, has to be saved. If no listing is needed, then the intermediate form can be reduced to the bare essentials.

Either way, another task of pass one is to save all macro definitions and expand the calls as they are encountered. Thus defining the symbols and expanding the macros are generally combined into one pass.

7.3.2 Pass One

The principal function of pass one is to build up a table called the **symbol table**, containing the values of all symbols. A symbol is either a label or a value that is assigned a symbolic name by means of a pseudoinstruction such as

```
BUFSIZE EQU 8192
```

In assigning a value to a symbol in the label field of an instruction, the assembler must know what address that instruction will have during execution of the program. To keep track of the execution-time address of the instruction being assembled, the assembler maintains a variable during assembly, known as the **ILC** (**Instruction Location Counter**). This variable is set to 0 at the beginning of pass one and incremented by the instruction length for each instruction processed, as shown in Fig. 7-6. This example is for the x86.

Label	Opcode	Operands	Comments	Length	ILC
MARIA:	MOV	EAX, I	EAX = I	5	100
	MOV	EBX, J	EBX = J	6	105
ROBERTA:	MOV	ECX, K	ECX = K	6	111
	IMUL	EAX, EAX	EAX = I * I	2	117
	IMUL	EBX, EBX	EBX = J * J	3	119
	IMUL	ECX, ECX	ECX = K * K	3	122
MARILYN:	ADD	EAX, EBX	EAX = I * I + J * J	2	125
	ADD	EAX, ECX	EAX = I * I + J * J + K * K	2	127
STEPHANY:	JMP	DONE	branch to DONE	5	129

Figure 7-6. The instruction location counter (ILC) keeps track of the address where the instructions will be loaded in memory. In this example, the statements prior to MARIA occupy 100 bytes.

Pass one of most assemblers uses at least three internal tables: the symbol table, the pseudoinstruction table, and the opcode table. If needed, a literal table is also kept. The symbol table has one entry for each symbol, as illustrated in Fig. 7-7. Symbols are defined either by using them as labels or by explicit definition (e.g., EQU). Each symbol table entry contains the symbol itself (or a pointer to it), its numerical value, and sometimes other information. This additional information may include

1. The length of data field associated with symbol.
2. The relocation bits. (Does the symbol change value if the program is loaded at a different address than the assembler assumed?)
3. Whether or not the symbol is to be accessible outside the procedure.

Symbol	Value	Other information
MARIA	100	
ROBERTA	111	
MARILYN	125	
STEPHANY	129	

Figure 7-7. A symbol table for the program of Fig. 7-6.

The opcode table contains at least one entry for each symbolic opcode (mnemonic) in the assembly language. Figure 7-8 shows part of an opcode table. Each entry contains the symbolic opcode, two operands, the opcode's numerical value, the instruction length, and a type number that separates the opcodes into groups depending on the number and kind of operands.

Opcode	First operand	Second operand	Hex opcode	Instruction length	Instruction class
AAA	—	—	37	1	6
ADD	EAX	immed32	05	5	4
ADD	reg	reg	01	2	19
AND	EAX	immed32	25	5	4
AND	reg	reg	21	2	19

Figure 7-8. A few excerpts from the opcode table for an x86 assembler.

As an example, consider the opcode ADD. If an ADD instruction contains EAX as the first operand and a 32-bit constant (immed32) as the second one, then opcode 0x05 is used and the instruction length is 5 bytes. (Constants that can be expressed in 8 or 16 bits use different opcodes, not shown.) If ADD is used with two registers as operands, the instruction is 2 bytes, with opcode 0x01. The (arbitrary) instruction class 19 would be given to all opcode-operand combinations that follow the same rules and should be processed the same way as ADD with two register operands. The instruction class effectively designates a procedure within the assembler that is called to process all instructions of a given type.

Some assemblers allow programmers to write instructions using immediate addressing even though no corresponding target language instruction exists. Such "pseudoimmediate" instructions are handled as follows. The assembler allocates

memory for the immediate operand at the end of the program and generates an instruction that references it. For instance, the IBM 360 mainframe and its successors have no immediate instructions. Nevertheless, programmers may write

```
L 14,=F'5'
```

to load register 14 with a full word constant 5. In this manner, the programmer avoids explicitly writing a pseudoinstruction to allocate a word initialized to 5, giving it a label, and then using that label in the L instruction. Constants for which the assembler automatically reserves memory are called **literals**. In addition to saving the programmer a little writing, literals improve the readability of a program by making the value of the constant apparent in the source statement. Pass one of the assembler must build a table of all literals used in the program. All three of our example computers have immediate instructions, so their assemblers do not provide literals. Immediate instructions are quite common nowadays, but formerly they were unusual. It is likely that the widespread use of literals made it clear to machine designers that immediate addressing was a good idea. If literals are needed, a literal table is maintained during assembly, with a new entry made each time a literal is encountered. After the first pass, this table is sorted and duplicates removed.

Figure 7-9 shows a procedure that could serve as a basis for pass one of an assembler. The style of programming is noteworthy in itself. The procedure names have been chosen to give a good indication of what the procedures do. Most important, Fig. 7-9 represents an outline of pass one which, although not complete, forms a good starting point. It is short enough to be easily understood and it makes clear what the next step must be—namely, to write the procedures used in it.

Some of these procedures will be relatively short, such as *check_for_symbol*, which just returns the symbol as a character string if there is one and null if there is not. Other procedures, such as *get_length_of_type1* and *get_length_of_type2*, may be longer and may call other procedures. In general, the number of types will not be two, of course, but will depend on the language being assembled and how many types of instructions it has.

Structuring programs in this way has other advantages in addition to ease of programming. If the assembler is being written by a group of people, the various procedures can be parceled out among the programmers. All the (nasty) details of getting the input are hidden away in *read_next_line*. If they should change—for example, due to an operating system change—only one subsidiary procedure is affected, and no changes are needed to the *pass_one* procedure itself.

As it reads the program, pass one of the assembler has to parse each line to find the opcode (e.g., ADD), look up its type (basically, the pattern of operands), and compute the instruction's length. This information is also needed on the second pass, so it is possible to write it out explicitly to eliminate the need to parse the line from scratch next time. However, rewriting the input file causes more I/O to

```
public static void pass_one( ) {
   // This procedure is an outline of pass one of a simple assembler.
   boolean more_input = true;                  // flag that stops pass one
   String line, symbol, literal, opcode;       // fields of the instruction
   int location_counter, length, value, type;  // misc. variables
   final int END_STATEMENT = –2;               // signals end of input

   location_counter = 0;                       // assemble first instruction at 0
   initialize_tables( );                       // general initialization

   while (more_input) {                        // more_input set to false by END
      line = read_next_line( );                // get a line of input
      length = 0;                              // # bytes in the instruction
      type = 0;                                // which type (format) is the instruction

      if (line_is_not_comment(line)) {
         symbol = check_for_symbol(line);      // is this line labeled?
         if (symbol != null)                   // if it is, record symbol and value
            enter_new_symbol(symbol, location_counter);
         literal = check_for_literal(line);    // does line contain a literal?
         if (literal != null)                  // if it does, enter it in table
            enter_new_literal(literal);

         // Now determine the opcode type.  –1 means illegal opcode.
         opcode = extract_opcode(line);        // locate opcode mnemonic
         type = search_opcode_table(opcode); // find format, e.g. OP REG1,REG2
         if (type < 0)                         // if not an opcode, is it a pseudoinstruction?
            type = search_pseudo_table(opcode);
         switch(type) {                        // determine the length of this instruction
            case 1: length = get_length_of_type1(line);  break;
            case 2: length = get_length_of_type2(line);  break;
            // other cases here
         }
      }

      write_temp_file(type, opcode, length, line);    // useful info for pass two
      location_counter = location_counter + length;       // update loc_ctr
      if (type == END_STATEMENT) {             // are we done with input?
         more_input = false;                   // if so, perform housekeeping tasks
         rewind_temp_for_pass_two( );          // like rewinding the temp file
         sort_literal_table( );                // and sorting the literal table
         remove_redundant_literals( );         // and removing duplicates from it
      }
   }
}
```

Figure 7-9. Pass one of a simple assembler.

occur. Whether it is better to do more I/O to eliminate parsing or less I/O and more parsing depends on the relative speed of the CPU and disk, the efficiency of the file system, and other factors. In this example we will write out a temporary file containing the type, opcode, length, and actual input line. It is this line that pass two reads instead of the raw input file.

When the END pseudoinstruction is read, pass one is over. The symbol table and literal tables can be sorted at this point if needed. The sorted literal table can be checked for duplicate entries, which can be removed.

7.3.3 Pass Two

The function of pass two is to generate the object program and possibly print the assembly listing. In addition, pass two must output certain information needed by the linker for linking up procedures assembled at different times into a single executable file. Figure 7-10 shows a sketch of a procedure for pass two.

```
public static void pass_two( ) {
   // This procedure is an outline of pass two of a simple assembler.
   boolean more_input = true;                  // flag that stops pass two
   String line, opcode;                        // fields of the instruction
   int location_counter, length, type;         // misc. variables
   final int END_STATEMENT = –2;               // signals end of input
   final int MAX_CODE = 16;                    // max bytes of code per instruction
   byte code[ ] = new byte[MAX_CODE];          // holds generated code per instruction

   location_counter = 0;                       // assemble first instruction at 0

   while (more_input) {                        // more_input set to false by END
      type = read_type( );                     // get type field of next line
      opcode = read_opcode( );                 // get opcode field of next line
      length = read_length( );                 // get length field of next line
      line = read_line( );                     // get the actual line of input

      if (type != 0) {                         // type 0 is for comment lines
         switch(type) {                        // generate the output code
            case 1: eval_type1(opcode, length, line, code);  break;
            case 2: eval_type2(opcode, length, line, code);  break;
            // other cases here
         }
      }

      write_output(code);                      // write the binary code
      write_listing(code, line);               // print one line on the listing
      location_counter = location_counter + length;      // update loc_ctr
      if (type == END_STATEMENT) {             // are we done with input?
         more_input = false;                   // if so, perform housekeeping tasks
         finish_up( );                         // odds and ends
      }
   }
}
```

Figure 7-10. Pass two of a simple assembler.

The operation of pass two is more-or-less similar to that of pass one: it reads the lines one at a time and processes them one at a time. Since we have written the type, opcode, and length at the start of each line (on the temporary file), all of these

are read in to save some parsing. The main work of the code generation is done by the procedures *eval_type1*, *eval_type2*, and so on. Each one handles a particular pattern, such as an opcode and two register operands. It generates the binary code for the instruction and returns it in *code*. Then it is written out. More likely, *write_output* just buffers the accumulated binary code and writes the file to disk in large chunks to reduce disk traffic.

The original source statement and the object code generated from it (in hexadecimal) can then be printed or put into a buffer for later printing. After the ILC has been adjusted, the next statement is fetched.

Up until now it has been assumed that the source program does not contain any errors. Anyone who has ever written a program, in any language, knows how realistic that assumption is. Some of the common errors are as follows:

1. A symbol has been used but not defined.
2. A symbol has been defined more than once.
3. The name in the opcode field is not a legal opcode.
4. An opcode is not supplied with enough operands.
5. An opcode is supplied with too many operands.
6. A number contains an invalid character like 143G6.
7. Illegal register use (e.g., a branch to a register).
8. The END statement is missing.

Programmers are quite ingenious at thinking up new kinds of errors to make. Undefined symbol errors are frequently caused by typing errors, so a clever assembler could try to figure out which of the defined symbols most resembles the undefined one and use that instead. Little can be done about correcting most other errors. The best thing for the assembler to do with an errant statement is to print an error message and try to continue assembly.

7.3.4 The Symbol Table

During pass one of the assembly process, the assembler accumulates information about symbols and their values that must be stored in the symbol table for lookup during pass two. Several different ways are available for organizing the symbol table. We will briefly describe some of them below. All of them attempt to simulate an **associative memory**, which conceptually is a set of (symbol, value) pairs. Given the symbol, the associative memory must produce the value.

The simplest implementation technique is indeed to implement the symbol table as an array of pairs, the first element of which is (or points to) the symbol and the second of which is (or points to) the value. Given a symbol to look up, the

symbol table routine just searches the table linearly until it finds a match. This method is easy to program but is slow, because, on the average, half the table will have to be searched on each lookup.

Another way to organize the symbol table is to sort it on the symbols and use the **binary search** algorithm to look up a symbol. This algorithm works by comparing the middle entry in the table to the symbol. If the symbol comes before the middle entry alphabetically, the symbol must be located in the first half of the table. If the symbol comes after the middle entry, it must be in the second half of the table. If the symbol is equal to the middle entry, the search terminates.

Assuming that the middle entry is not equal to the symbol sought, we at least know which half of the table to look for it in. Binary search can now be applied to the correct half, which yields either a match, or the correct quarter of the table. Applying the algorithm recursively, a table of size n entries can be searched in about $\log_2 n$ attempts. Obviously, this way is much faster than searching linearly, but it requires maintaining the table in sorted order.

A completely different way of simulating an associative memory is a technique known as **hash coding** or **hashing**. This approach requires having a "hash" function that maps symbols onto integers in the range 0 to $k - 1$. One possible function is to multiply the ASCII codes of the characters in the symbols together, ignoring overflow, and taking the result modulo k or dividing it by a prime number. In fact, almost any function of the input that gives a uniform distribution of the hash values will do. Symbols can be stored by having a table consisting of k **buckets** numbered 0 to $k - 1$. All the (symbol, value) pairs whose symbol hashes to i are stored on a linked list pointed to by slot i in the hash table. With n symbols and k slots in the hash table, the average list will have length n/k. By choosing k approximately equal to n, symbols can be located with only about one lookup on the average. By adjusting k we can reduce table size at the expense of slower lookups. Hash coding is illustrated in Fig. 7-11.

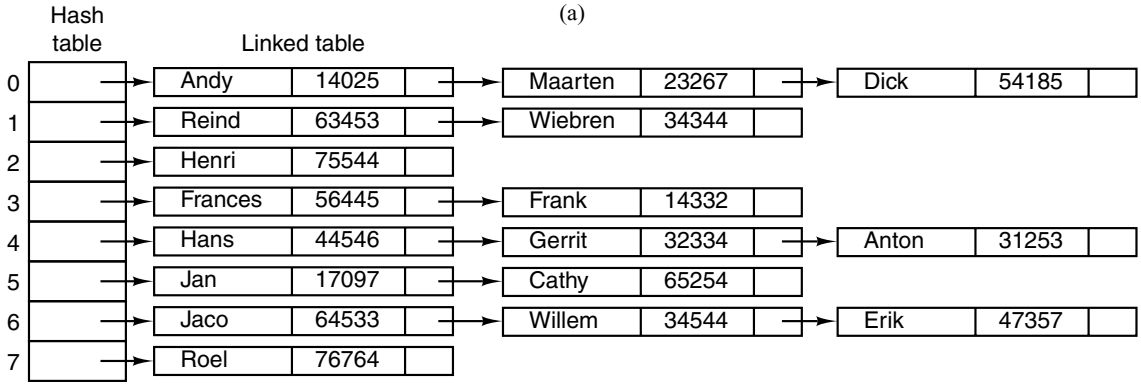
7.4 LINKING AND LOADING

Most programs consist of more than one procedure. Compilers and assemblers generally translate one procedure at a time and put the translated output on disk. Before the program can be run, all the translated procedures must be found and linked together properly. If virtual memory is not available, the linked program must be explicitly loaded into main memory as well. Programs that perform these functions are called by various names, including **linker**, **linking loader**, and **linkage editor**. The complete translation of a source program requires two steps, as shown in Fig. 7-12:

1. Compilation or assembly of the source files.
2. Linking of the object modules.

Andy	14025	0
Anton	31253	4
Cathy	65254	5
Dick	54185	0
Erik	47357	6
Frances	56445	3
Frank	14332	3
Gerrit	32334	4
Hans	44546	4
Henri	75544	2
Jan	17097	5
Jaco	64533	6
Maarten	23267	0
Reind	63453	1
Roel	76764	7
Willem	34544	6
Wiebren	34344	1

(a)



(b)

Figure 7-11. Hash coding. (a) Symbols, values, and the hash codes derived from the symbols. (b) Eight-entry hash table with linked lists of symbols and values.

The first step is performed by the compiler or assembler and the second one is performed by the linker.

The translation from source procedure to object module represents a change of level because the source language and target language have different instructions and notation. The linking process, however, does not represent a change of level, since both the linker's input and the linker's output are programs for the same virtual machine. The linker's function is to collect procedures translated separately and link them together to be run as a unit called an **executable binary program**. On Windows systems, the object modules have extension *.obj* and the executable binary programs have extension *.exe*. On UNIX, the object modules have extension *.o*; executable binary programs have no extension.

Compilers and assemblers translate each source file separately for a very good reason. If a compiler or assembler were to read a series of source procedures and immediately produce a ready-to-run machine language program, changing one

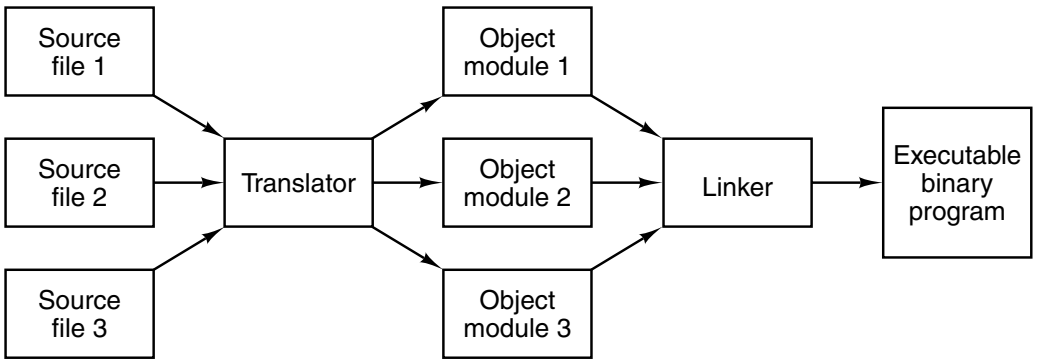


Figure 7-12. Generation of an executable binary program from a collection of independently translated source procedures requires using a linker.

statement in one source procedure would require all the procedures to be retranslated.

If the separate-object-module technique of Fig. 7-12 is used, it is only necessary to retranslate the modified procedure and not the unchanged ones, although it is necessary to relink all the object modules again. Linking is usually much faster than translating, however; thus the two-step process of translating and linking can save a great deal of time during the development of a program. This gain is especially important for programs with hundreds or thousands of modules.

7.4.1 Tasks Performed by the Linker

At the start of pass one of the assembly process, the instruction location counter is set to 0. This step is equivalent to assuming that the object module will be located at (virtual) address 0 during execution. Figure 7-13 shows four object modules for a generic machine. In this example, each module begins with a BRANCH instruction to a MOVE instruction within the module.

In order to run the program, the linker brings the object modules from the disk into main memory to form the image of the executable binary program, as shown in Fig. 7-14(a). The idea is to make an exact image of the executable program's virtual address space inside the linker and position all the object modules at their correct locations. If there is not enough (virtual) memory to form the image, a disk file can be used. Typically, a small section of memory starting at address zero is used for interrupt vectors, communication with the operating system, catching uninitialized pointers, or other purposes, so programs often start above 0. In this figure we have (arbitrarily) started programs at address 100.

The program of Fig. 7-14(a), although loaded into the image of the executable binary file, is not yet ready for execution. Consider what would happen if execution began with the instruction at the beginning of module *A*. The program would not branch to the MOVE instruction as it should, because that instruction is now at

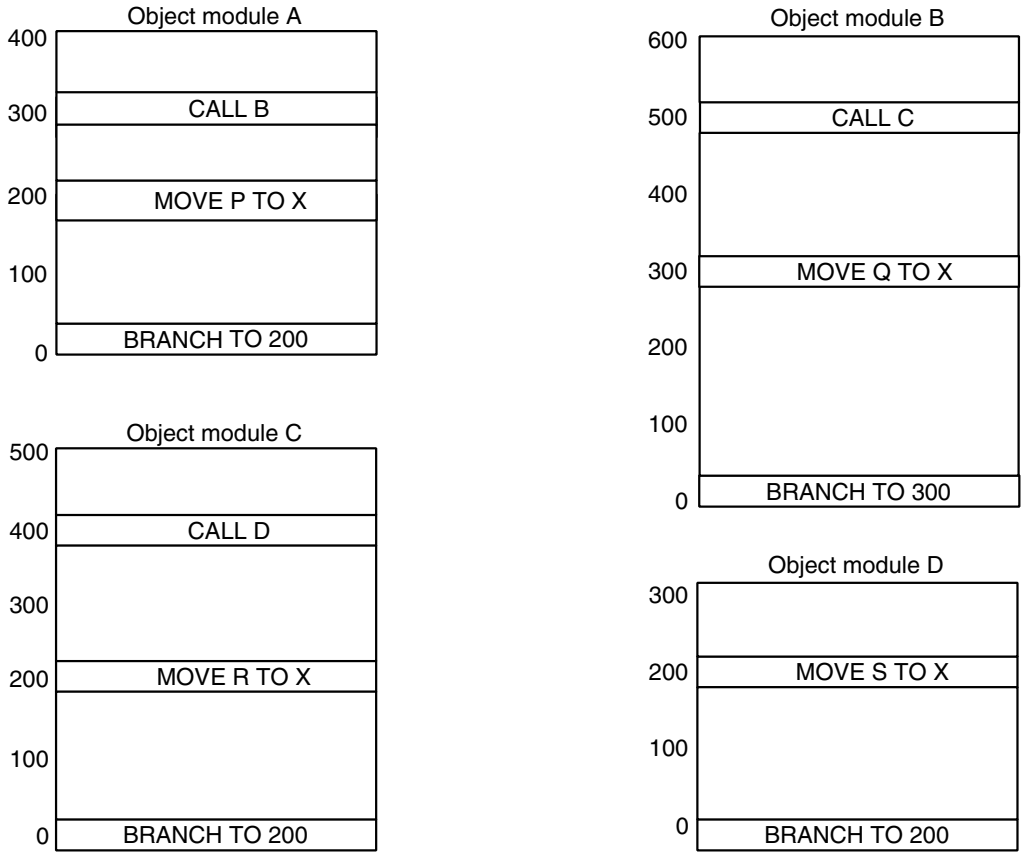


Figure 7-13. Each module has its own address space, starting at 0.

300. In fact, all memory reference instructions will fail for the same reason. Clearly something has to be done.

This problem, called the **relocation problem**, occurs because each object module in Fig. 7-13 represents a separate address space. On a machine with a segmented address space, such as the x86, theoretically each object module could have its own address space by being placed in its own segment. However, OS/2 was the only operating system for the x86 that supported this concept. All versions of Windows and UNIX support only one linear address space, so all the object modules must be merged together into a single address space.

Furthermore, the procedure call instructions in Fig. 7-14(a) will not work either. At address 400, the programmer had intended to call object module *B*, but because each procedure is translated by itself, the assembler has no way of knowing what address to insert into the CALL B instruction. The address of object module *B* is not known until linking time. This problem is called the **external reference** problem. Both of these problems can be solved in a simple way by the linker.

The linker merges the separate address spaces of the object modules into a single linear address space in the following steps:

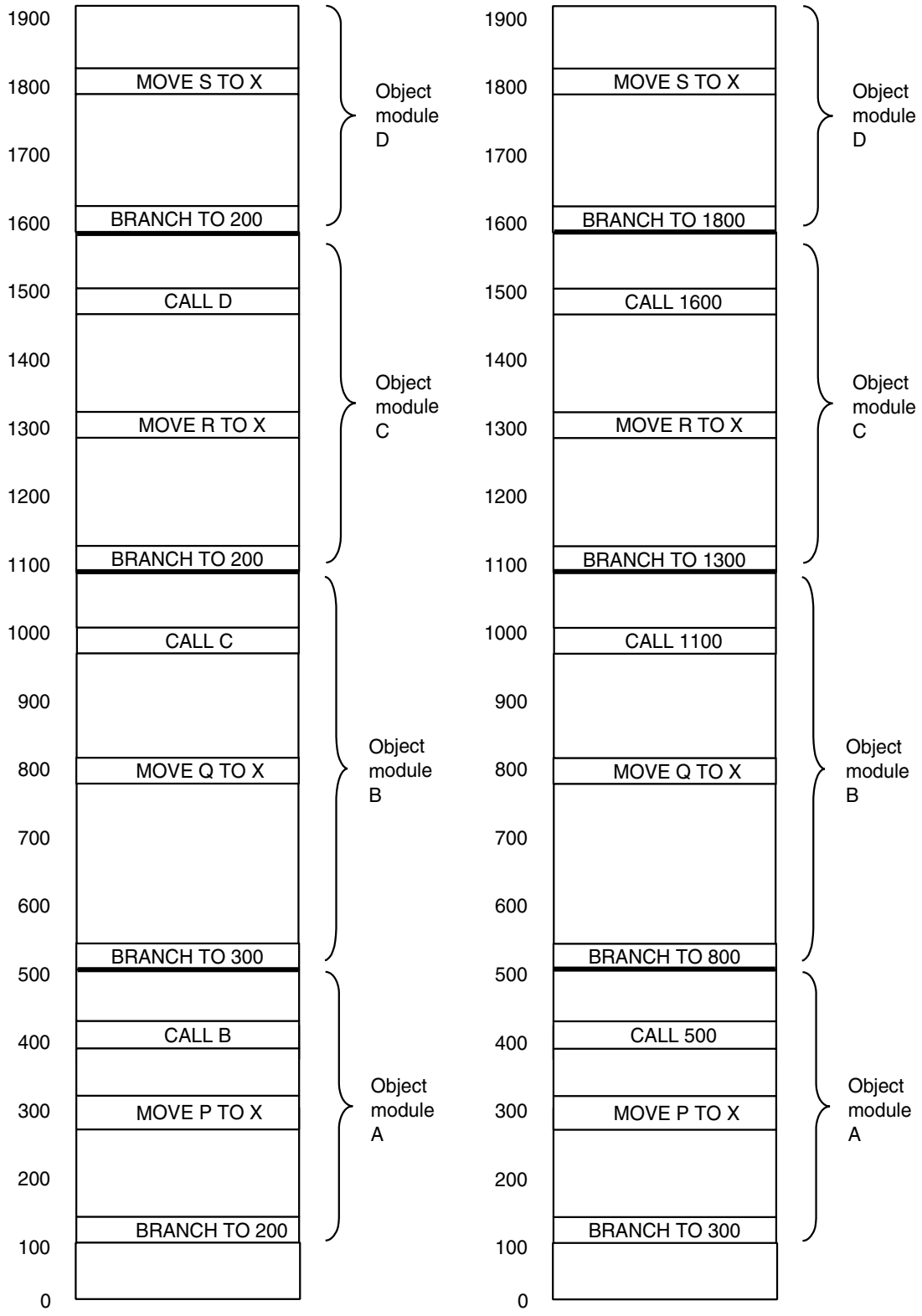


Figure 7-14. (a) The object modules of Fig. 7-13 after being positioned in the binary image but before being relocated and linked. (b) The same object modules after linking and after relocation has been performed.

1. It constructs a table of all the object modules and their lengths.
2. Based on this table, it assigns a base address to each object module.
3. It finds all the instructions that reference memory and adds to each a **relocation constant** equal to the starting address of its module.
4. It finds all the instructions that reference other procedures and inserts the address of these procedures in place.

The object module table constructed in step 1 is shown for the modules of Fig. 7-14 below. It gives the name, length, and starting address of each module.

Module	Length	Starting address
A	400	100
B	600	500
C	500	1100
D	300	1600

Figure 7-14(b) shows how the address space of Fig. 7-14(a) looks after the linker has performed these steps.

7.4.2 Structure of an Object Module

Object modules often contain six parts, as shown in Fig. 7-15. The first part contains the name of the module, certain information needed by the linker, such as the lengths of the various parts of the module, and sometimes the assembly date.

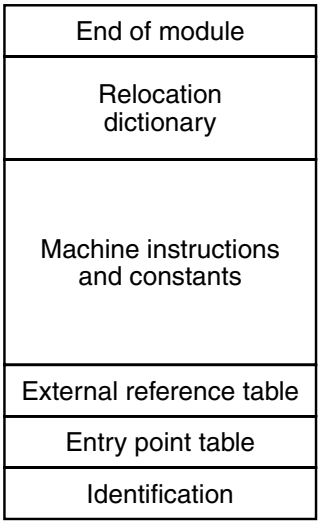


Figure 7-15. The internal structure of an object module produced by a translator. The *Identification* field comes first.

The second part of the object module is a list of the symbols defined in the module that other modules may reference, together with their values. For example, if the module consists of a procedure named *bigbug*, the entry point table will

contain the character string "bigbug" followed by the address to which it corresponds. The assembly language programmer indicates which symbols are to be declared as **entry points** by using a pseudoinstruction such as PUBLIC in Fig. 7-2.

The third part of the object module consists of a list of the symbols that are used in the module but which are defined in other modules, along with a list of which machine instructions use which symbols. The linker needs the latter list in order to be able to insert the correct addresses into the instructions that use external symbols. A procedure can call other independently translated procedures by declaring the names of the called procedures to be external. The assembly language programmer indicates which symbols are to be declared as **external symbols** by using a pseudoinstruction such as EXTERN in Fig. 7-2. On some computers entry points and external references are combined into one table.

The fourth part of the object module is the assembled code and constants. This part of the object module is the only one that will be loaded into memory to be executed. The other five parts will be used by the linker to help it do its work and then discarded before execution begins.

The fifth part of the object module is the relocation dictionary. As shown in Fig. 7-14, instructions that contain memory addresses must have a relocation constant added. Since the linker has no way of telling by inspection which of the data words in part four contain machine instructions and which contain constants, information about which addresses are to be relocated is provided in this table. The information may take the form of a bit table, with 1 bit per potentially relocatable address, or an explicit list of addresses to be relocated.

The sixth part is an end-of-module mark, perhaps a checksum to catch errors made while reading the module, and the address at which to begin execution.

Most linkers require two passes. On pass one the linker reads all the object modules and builds up a table of module names and lengths, and a global symbol table consisting of all entry points and external references. On pass two the object modules are read, relocated, and linked one module at a time.

7.4.3 Binding Time and Dynamic Relocation

In a multiprogramming system, a program can be read into main memory, run for a little while, written to disk, and then read back into main memory to be run again. In a large system, with many programs, it is difficult to ensure that a program is read back into the same locations every time.

Figure 7-16 shows what would happen if the already relocated program of Fig. 7-14(b) were reloaded at address 400 instead of address 100 where the linker put it originally. All the memory addresses are incorrect; moreover, the relocation information has long since been discarded. Even if the relocation information were still available, the cost of having to relocate all the addresses every time the program was swapped would be too high.

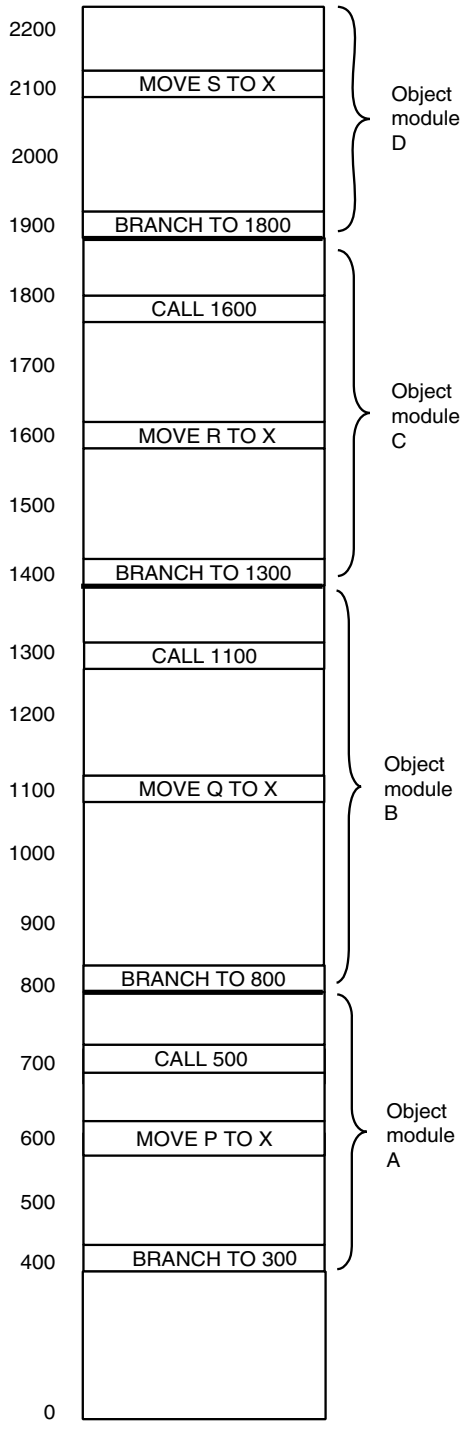


Figure 7-16. The relocated binary program of Fig. 7-14(b) moved up 300 addresses. Many instructions now refer to an incorrect memory address.

The problem of moving programs that have already been linked and relocated is intimately related to the time at which the final binding of symbolic names onto absolute physical memory addresses is completed. When a program is written it contains symbolic names for memory addresses, for example, `BR L`. The time at which the actual main memory address corresponding to *L* is determined is called the **binding time**. At least six possibilities for the binding time exist:

1. When the program is written.
2. When the program is translated.
3. When the program is linked but before it is loaded.
4. When the program is loaded.
5. When a base register used for addressing is loaded.
6. When the instruction containing the address is executed.

If an instruction containing a memory address is moved after binding, it will be incorrect (assuming that the object referred to has also been moved). If the translator produces an executable binary as output, the binding has occurred at translation time, and the program must be run at the address at which the translator expected it to be run at. The linking method described in the preceding section binds symbolic names to absolute addresses during linking, which is why moving programs after linking fails, as shown in Fig. 7-16.

Two related issues are involved here. First, there is the question of when symbolic names are bound to virtual addresses. Second, there is a question of when virtual addresses are bound to physical addresses. Only when both operations have taken place is binding complete. When the linker merges the separate address spaces of the object modules into a single linear address space, it is, in fact, creating a virtual address space. The relocation and linking serve to bind symbolic names onto specific virtual addresses. This observation is true whether or not virtual memory is being used.

Assume for the moment that the address space of Fig. 7-14(b) is paged. It is clear that the virtual addresses corresponding to the symbolic names *A*, *B*, *C*, and *D* have already been determined, even though their physical main memory addresses will depend on the contents of the page table at the time they are used. An executable binary program is really a binding of symbolic names to virtual addresses.

Any mechanism that allows the mapping of virtual addresses onto physical memory addresses to be changed easily will facilitate moving programs around in main memory, even after they have been bound to a virtual address space. One such mechanism is paging. After a program has been moved in main memory, only its page table need be changed, not the program itself.

A second mechanism is the use of a runtime relocation register. The CDC 6600 and its successors had such a register. On machines using this relocation

technique, the register always points to the physical address of the start of the current program. All memory addresses have the contents of the relocation register added to them by the hardware before being sent to the memory. The entire relocation process is transparent to the user programs. They do not even know that it is occurring. When a program is moved, the operating system must update the relocation register. This mechanism is less general than paging because the entire program must be moved as a unit (unless there are separate code and data relocation registers, as on the Intel 8088, in which case it has to be moved as two units).

A third mechanism is possible on machines that can refer to memory relative to the program counter. Many branch instructions are relative to the program counter, which helps. Whenever a program is moved in main memory only the program counter need be updated. A program, all of whose memory references are either relative to the program counter or absolute (e.g., to I/O device registers at absolute addresses) is said to be **position independent**. A position-independent procedure can be placed anywhere within the virtual address space without the need for relocation.

7.4.4 Dynamic Linking

The linking strategy discussed in Sec. 7.4.1 has the property that all procedures that a program might call are linked before the program can begin execution. On a computer with virtual memory, completing all linking before beginning execution does not take advantage of the full capabilities of the virtual memory. Many programs have procedures that are called only under unusual circumstances. For example, compilers have procedures for compiling rarely used statements, plus procedures for handling error conditions that seldom occur.

A more flexible way to link separately compiled procedures is to link each procedure at the time it is first called. This process is known as **dynamic linking**. It was pioneered by MULTICS whose implementation is in some ways still unsurpassed. In the next sections we will look at dynamic linking in several systems.

Dynamic Linking in MULTICS

In the MULTICS form of dynamic linking, associated with each program is a segment, called the **linkage segment**, which contains one block of information for each procedure that might be called. This block of information starts with a word reserved for the virtual address of the procedure and it is followed by the procedure name, which is stored as a character string.

When dynamic linking is being used, procedure calls in the source language are translated into instructions that indirectly address the first word of the corresponding linkage block, as shown in Fig. 7-17(a). The compiler fills this word with either an invalid address or a special bit pattern that forces a trap.

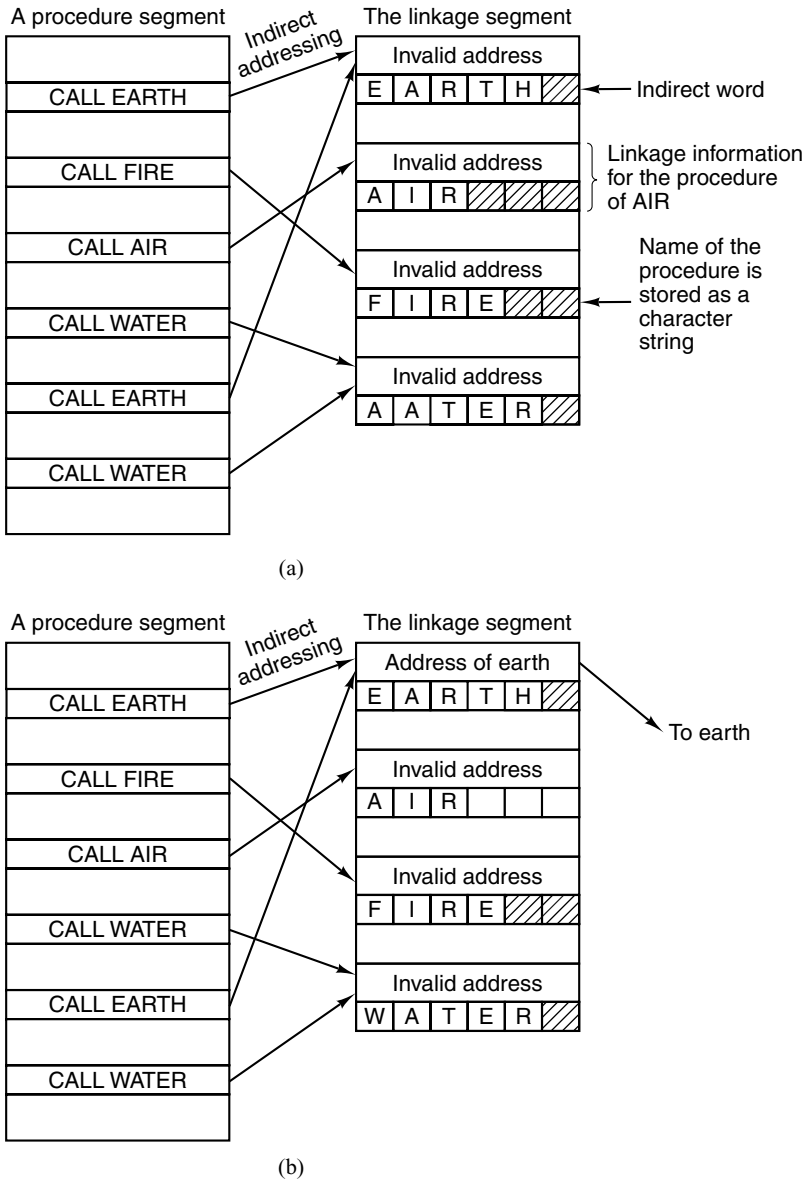


Figure 7-17. Dynamic linking. (a) Before *EARTH* is called. (b) After *EARTH* has been called and linked.

When a procedure in a different segment is called, the attempt to address the invalid word indirectly causes a trap to the dynamic linker. The linker then finds the character string in the word following the invalid address and searches the user's file directory for a compiled procedure with this name. That procedure is then assigned a virtual address, usually in its own private segment, and this virtual address overwrites the invalid address in the linkage segment, as indicated in Fig. 7-17(b). Next, the instruction causing the linkage fault is re-executed, allowing the program to continue from the place it was before the trap.

All subsequent references to that procedure will be executed without causing a linkage fault, for the indirect word now contains a valid virtual address. Consequently, the dynamic linker is invoked only the first time a procedure is called.

Dynamic Linking in Windows

All versions of the Windows operating system support dynamic linking and rely heavily on it. Dynamic linking uses a special file format called a **DLL** (**Dynamic Link Library**). DLLs can contain procedures, data, or both. They are commonly used to allow two or more processes to share library procedures or data. Many DLLs have extension *.dll*, but other extensions are also in use, including *.drv* (for driver libraries) and *.fon* (for font libraries).

The most common form of a DLL is a library consisting of a collection of procedures that can be loaded into memory and accessed by multiple processes at the same time. Figure 7-18 illustrates two processes sharing a DLL file that contains four procedures, *A*, *B*, *C*, and *D*. Program 1 uses procedure *A*; program 2 uses procedure *C*, although they could equally well have used the same procedure.

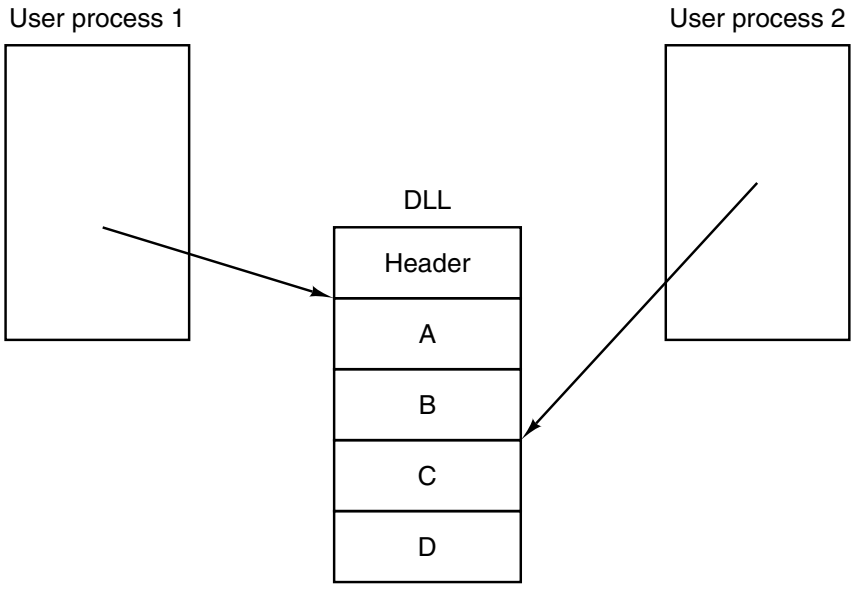


Figure 7-18. Use of a DLL file by two processes.

A DLL is constructed by the linker from a collection of input files. In fact, building a DLL file is very much like building an executable binary program, except that a special flag is given to the linker to tell it to make a DLL. DLLs are commonly constructed from collections of library procedures that are likely to be needed by multiple processes. The interface procedures to the Windows system call library and large graphics libraries are common examples of DLLs. The advantage o using DLLs is saving space in memory and on disk. If some common li-

brary were statically bound to each program using it, it would appear in many executable binaries on the disk and in memory, wasting space. With DLLs, each library appears only once on disk and once in memory.

In addition to saving space, this approach makes it easy to update library procedures, even after the programs using them have been compiled and linked. For commercial software packages, where the users rarely have the source code, using DLLs means that the software vendor can fix bugs in the libraries by just distributing new DLL files over the Internet, without requiring any changes to the main program binaries.

The main difference between a DLL and an executable binary is that a DLL cannot be started and run on its own (because it has no main program). It also has different information in its header. In addition, the DLL as a whole has several extra procedures not related to the procedures in the library. For example, there is one procedure that is automatically called whenever a new process is bound to the DLL and another one that is automatically called whenever a process is unbound from it. These procedures can allocate and deallocate memory or manage other resources needed by the DLL.

There are two ways for a program to bind to a DLL. In the first way, called **implicit linking**, the user's program is statically linked with a special file called an **import library** that is generated by a utility program that extracts certain information from the DLL. The import library provides the glue that allows the user program to access the DLL. A user program can be linked with multiple import libraries. When a program using implicit linking is loaded into memory for execution, Windows examines it to see which DLLs it uses and checks to see if all of them are already in memory. Those that are not in memory are loaded immediately (but not necessarily in their entirety, since they are paged). Some changes are then made to the data structures in the import libraries so the called procedures can be located, somewhat analogous to the changes shown in Fig. 7-17. They also have to be mapped into the program's virtual address space. At this point, the user program is ready to run and can call the procedures in the DLLs as though they had been statically bound with it.

The alternative to implicit linking is (not surprisingly) **explicit linking**. This approach does not require import libraries and does not cause the DLLs to be loaded at the same time the user program is. Instead, the user program makes an explicit call at run time to bind to a DLL, then makes additional calls to get the addresses of procedures it needs. Once these have been found, it can call the procedures. When it is all done, it makes a final call to unbind from the DLL. When the last process unbinds from a DLL, the DLL can be removed from memory.

It is important to realize that a procedure in a DLL does not have any identity of its own (as a thread or process does). It runs in the called's thread and uses the called's stack for its local variables. It can have process-specific static data (as well as shared data) and otherwise behaves the same as a statically-linked procedure. The only essential difference is how the binding to it is performed.

Dynamic Linking in UNIX

The UNIX system has a mechanism similar in essence to DLLs in Windows. It is called a **shared library**. Like a DLL file, a shared library is an archive file containing multiple procedures or data modules that are present in memory at run time and can be bound to multiple processes at the same time. The standard C library and much of the networking code are shared libraries.

UNIX supports only implicit linking, so a shared library consist of two parts: a **host library**, which is statically linked with the executable file, and a **target library**, which is called at run time. While the details differ, the concepts are essentially the same as with DLLs.

7.5 SUMMARY

Although most programs can and should be written in a high-level language, occasional situations exist in which assembly language is needed, at least in part. Programs for resource-poor computers such as smart cards and embedded processors in small consumer devices like clock radios are potential candidates. An assembly language program is a symbolic representation for some underlying machine language program. It is translated to the machine language by a program called an assembler.

When extremely fast execution is critical to the success of some application, a better approach than writing everything in assembly language is to first write the whole program in a high-level language, then measure where it is spending its time, and finally rewrite only those portions of the program that are heavily used. In practice, a small fraction of the code is usually responsible for a large fraction of the execution time.

Many assemblers have a macro facility that allows the programmer to give commonly used code sequences symbolic names for subsequent inclusion. Usually, these macros can be parameterized in a straightforward way. Macros are implemented by a kind of literal string-processing algorithm.

Most assemblers are two pass. Pass one is devoted to building up a symbol table for labels, literals, and explicitly declared identifiers. The symbols can either be kept unsorted and then searched linearly, first sorted and then searched using binary search, or hashed. If symbols do not need to be deleted during pass one, hashing is usually the best method. Pass two does the code generation. Some pseudoinstructions are carried out on pass one and some on pass two.

Independently-assembled programs can be linked together to form an executable binary program that can be run. This work is done by the linker. Its primary tasks are relocation and binding of names. Dynamic linking is a technique in which certain procedures are not linked until they are actually called. Windows DLLs and UNIX shared libraries use dynamic linking.

PROBLEMS

1. For a certain program, 2% of the code accounts for 50% of the execution time. Compare the following three strategies with respect to programming time and execution time. Assume that it would take 100 man-months to write it in C, and that assembly code is 10 times slower to write and four times more efficient.

a. Entire program in C.
b. Entire program in assembler.
c. First all in C, then the key 2% rewritten in assembler.

2. Do the considerations that hold for two-pass assemblers also hold for compilers?

a. Assume that the compilers produce object modules, not assembly code.
b. Assume that the compilers produce symbolic assembly language.

3. Most assemblers for the x86 have the destination address as the first operand and the source address as the second operand. What problems would have to be solved to do it the other way?

4. Can the following program be assembled in two passes? EQU is a pseudoinstruction that equates the label to the expression in the operand field.

```
P EQU Q
Q EQU R
R EQU S
S EQU 4
```

5. The Dirtcheap Software Company is planning to produce an assembler for a computer with a 48-bit word. To keep costs down, the project manager, Dr. Scrooge, has decided to limit the length of allowed symbols so that each symbol can be stored in a single word. Scrooge has declared that symbols may consist only of letters, except the letter Q, which is forbidden (to demonstrate their concern for efficiency to the customers). What is the maximum length of a symbol? Describe your encoding scheme.

6. What is the difference between an instruction and a pseudoinstruction?

7. What is the difference between the instruction location counter and the program counter, if any? After all, both keep track of the next instruction in a program.

8. Show the symbol table after the following x86 statements have been encountered. The first statement is assigned to address 1000.

```
EVEREST:    POP BX          (1 BYTE)
K2:         PUSH BP         (1 BYTE)
WHITNEY:    MOV BP,SP       (2 BYTES)
MCKINLEY:   PUSH X          (3 BYTES)
FUJI:       PUSH SI         (1 BYTE)
KIBO:       SUB SI,300      (3 BYTES)
```

9. Can you envision circumstances in which an assembly language permits a label to be the same as an opcode (e.g., *MOV* as a label)? Discuss.

10. Show the steps needed to look up Ann Arbor using binary search on the following list: Ann Arbor, Berkeley, Cambridge, Eugene, Madison, New Haven, Palo Alto, Pasadena, Santa Cruz, Stony Brook, Westwood, and Yellow Springs. When computing the middle element of a list with an even number of elements, use the element just after the middle index.

11. Is it possible to use binary search on a table whose size is prime?

12. Compute the hash code for each of the following symbols by adding up the letters (A = 1, B = 2, etc.) and taking the result modulo the hash table size. The hash table has 19 slots, numbered 0 to 18.

els, jan, jelle, maaike

Does each symbol generate a unique hash value? If not, how can the collision problem be dealt with?

13. The hash coding method described in the text links all the entries having the same hash code together on a linked list. An alternative method is to have only a single n-slot table, with each table slot having room for one key and its value (or pointers to them). If the hashing algorithm generates a slot that is already full, a second hashing algorithm is used to try again. If that one is also full, another is used, and so on, until an empty is found. If the fraction of the slots that are full is R, how many probes will be needed, on the average, to enter a new symbol?

14. As technology progresses, it may one day be possible to put thousands of identical CPUs on a chip, each CPU having a few words of local memory. If all CPUs can read and write three shared registers, how can an associative memory be implemented?

15. The x86 has a segmented architecture, with multiple independent segments. An assembler for this machine might well have a pseudoinstruction `SEG N` that would direct the assembler to place subsequent code and data in segment N. Does this scheme have any influence on the ILC?

16. Programs often link to multiple DLLs. Would it not be more efficient just to put all the procedures in one big DLL and then link to it?

17. Can a DLL be mapped into two process' virtual address spaces at different virtual addresses? If so, what problems arise? Can they be solved? If not, what can be done to eliminate them?

18. One way to do (static) linking is as follows. Before scanning the library, the linker builds a list of procedures needed, that is, names defined as `EXTERN` in the modules being linked. Then the linker goes through the library linearly, extracting every procedure that is in the list of names needed. Does this scheme work? If not, why not and how can it be remedied?

19. Can a register be used as the actual parameter in a macro call? How about a constant? Why or why not?

20. You are to implement a macro assembler. For esthetic reasons, your boss has decided that macro definitions need not precede their calls. What implications does this decision have on the implementation?

21. Think of a way to put a macro assembler into an infinite loop.

22. A linker reads five modules, whose lengths are 200, 800, 600, 500, and 700 words, respectively. If they are loaded in that order, what are the relocation constants?

23. Write a symbol table package consisting of two routines: *enter(symbol, value)* and *lookup(symbol, value)*. The former enters new symbols in the table and the latter looks them up. Use some form of hash coding.

24. Repeat the previous problem, only this time instead of using a hash table, after the last symbol is entered, sort the table and use a binary-lookup algorithm to find symbols.

25. Write a simple assembler for the Mic-1 computer of Chap. 4. In addition to handling the machine instructions, provide a facility for assigning constants to symbols at assembly time, and a way to assemble a constant into a machine word. These should be pseudoinstructions, of course.

26. Add a simple macro facility to the assembler of the preceding problem.