Appendix B

Assembly Language and Related Topics

The topic of assembly language was briefly introduced in Chapter 13. This appendix provides more detail and also covers a number of related topics. There are a number of reasons why it is worthwhile to study assembly language programming (as compared with programming in a higher-level language), including the following:

1. It clarifies the execution of instructions.
2. It shows how data are represented in memory.
3. It shows how a program interacts with the operating system, processor, and the I/O system.
4. It clarifies how a program accesses external devices.
5. Understanding assembly language programmers makes students better high-level language (HLL) programmers, by giving them a better idea of the target language that the HLL must be translated into.

We begin this chapter with a study of the basic elements of an assembly language, using the x86 architecture for our examples.[1] Next, we look at the operation of the assembler. This is followed by a discussion of linkers and loaders.

Table B.1 defines some of the key terms used in this appendix.

B.1 ASSEMBLY LANGUAGE

Assembly language is a programming language that is one step away from machine language. Typically, each assembly language instruction is translated into one machine instruction by the assembler. Assembly language is hardware dependent, with a different assembly language for each type of processor. In particular, assembly language instructions can make reference to specific registers in the processor, include all of the opcodes of the processor, and reflect the bit length of the various registers of the processor and operands of the machine language. An assembly language programmer must therefore understand the computer's architecture.

Programmers rarely use assembly language for applications or even systems programs. HLLs provide an expressive power and conciseness that greatly eases the programmer's tasks. The disadvantages of using an assembly language rather than an HLL include the following [FOG08]:

1. **Development time.** Writing code in assembly language takes much longer than writing in a high-level language.
2. **Reliability and security.** It is easy to make errors in assembly code. The assembler is not checking if the calling conventions and register save conventions are obeyed. Nobody is checking for you if the number of PUSH and POP instructions is the same in all possible branches and paths. There are so many possibilities for hidden errors in assembly code that it affects the reliability and security of the project unless you have a very systematic approach to testing and verifying.

[1]There are a number of assemblers for the x86 architecture. Our examples use NASM (Netwide Assembler), an open source assembler. A copy of the NASM manual is at this book's Premium Content site.

Table B.1 Key Terms for this Appendix

Assembler	A program that translates assembly language into machine code.
Assembly Language	A symbolic representation of the machine language of a specific processor, augmented by additional types of statements that facilitate program writing and that provide instructions to the assembler.
Compiler	A program that converts another program from some source language (or programming language) to machine language (object code). Some compilers output assembly language which is then converted to machine language by a separate assembler. A compiler is distinguished from an assembler by the fact that each input statement does not, in general, correspond to a single machine instruction or fixed sequence of instructions. A compiler may support such features as automatic allocation of variables, arbitrary arithmetic expressions, control structures such as FOR and WHILE loops, variable scope, input/output operations, higher-order functions and portability of source code.
Executable Code	The machine code generated by a source code language processor such as an assembler or compiler. This is software in a form that can be run in the computer.
Instruction Set	The collection of all possible instructions for a particular computer; that is, the collection of machine language instructions that a particular processor understands.
Linker	A utility program that combines one or more files containing object code from separately compiled program modules into a single file containing loadable or executable code.
Loader	A program routine that copies an executable program into memory for execution.
Machine Language, or Machine Code	The binary representation of a computer program which is actually read and interpreted by the computer. A program in machine code consists of a sequence of machine instructions (possibly interspersed with data). Instructions are binary strings which may be either all the same size (e.g., one 32-bit word for many modern RISC microprocessors) or of different sizes.
Object Code	The machine language representation of programming source code. Object code is created by a compiler or assembler and is then turned into executable code by the linker.

3. **Debugging and verifying.** Assembly code is more difficult to debug and verify because there are more possibilities for errors than in high-level code.
4. **Maintainability.** Assembly code is more difficult to modify and maintain because the language allows unstructured spaghetti code and all kinds of tricks that are difficult for others to understand. Thorough documentation and a consistent programming style are needed.
5. **Portability.** Assembly code is platform-specific. Porting to a different platform is difficult.

6. **System code can use intrinsic functions instead of assembly.** The best modern C++ compilers have intrinsic functions for accessing system control registers and other system instructions. Assembly code is no longer needed for device drivers and other system code when intrinsic functions are available.
7. **Application code can use intrinsic functions or vector classes instead of assembly.** The best modern C++ compilers have intrinsic functions for vector operations and other special instructions that previously required assembly programming.
8. **Compilers have been improved a lot in recent years.** The best compilers are now quite good. It takes a lot of expertise and experience to optimize better than the best C++ compiler.

Yet there are still some advantages to the occasional use of assembly language, including the following [FOG08a]:

1. **Debugging and verifying.** Looking at compiler-generated assembly code or the disassembly window in a debugger is useful for finding errors and for checking how well a compiler optimizes a particular piece of code.
2. **Making compilers.** Understanding assembly coding techniques is necessary for making compilers, debuggers, and other development tools.
3. **Embedded systems.** Small embedded systems have fewer resources than PCs and mainframes. Assembly programming can be necessary for optimizing code for speed or size in small embedded systems.
4. **Hardware drivers and system code.** Accessing hardware, system control registers, and so on may sometimes be difficult or impossible with high level code.
5. **Accessing instructions that are not accessible from high-level language.** Certain assembly instructions have no high-level language equivalent.
6. **Self-modifying code.** Self-modifying code is generally not profitable because it interferes with efficient code caching. It may, however, be advantageous, for example, to include a small compiler in math programs where a user-defined function has to be calculated many times.
7. **Optimizing code for size.** Storage space and memory is so cheap nowadays that it is not worth the effort to use assembly language for reducing code size. However, cache size is still such a critical resource that it may be useful in some cases to optimize a critical piece of code for size in order to make it fit into the code cache.
8. **Optimizing code for speed.** Modern C++ compilers generally optimize code quite well in most cases. But there are still cases where compilers perform poorly and where dramatic increases in speed can be achieved by careful assembly programming.
9. **Function libraries.** The total benefit of optimizing code is higher in function libraries that are used by many programmers.
10. **Making function libraries compatible with multiple compilers and operating systems.** It is possible to make library functions with multiple entries that are compatible with different compilers and different operating systems. This requires assembly programming.

The terms *assembly language* and *machine language* are sometimes, erroneously, used synonymously. Machine language consists of instructions directly executable by the processor. Each machine language instruction is a binary string containing an opcode, operand references, and perhaps other bits related to execution, such as flags. For convenience, instead of writing an instruction as a bit string, it can be written symbolically, with names for opcodes and registers. An assembly language makes much greater use of symbolic names, including assigning names to specific main memory locations and specific instruction locations. Assembly language also includes statements that are not directly executable but serve as instructions to the assembler that produces machine code from an assembly language program.

Assembly Language Elements

A statement in a typical assembly language has the form shown in Figure B.1. It consists of four elements: label, mnemonic, operand, and comment.

LABEL If a label is present, the assembler defines the label as equivalent to the address into which the first byte of the object code generated for that instruction will be loaded. The programmer may subsequently use the label as an address or as data in another instruction's address field. The assembler replaces the label with the assigned value when creating an object program. Labels are most frequently used in branch instructions.

As an example, here is a program fragment:

```
L2: SUB EAX, EDX   ;subtract contents of register EDX from
                   ;contents of EAX and store result in EAX
    JG  L2         ;jump to L2 if result of subtraction is
                   ;positive
```

The program will continue to loop back to location L2 until the result is zero or negative. Thus, when the jg instruction is executed, if the result is positive, the processor places the address equivalent to the label L2 in the program counter.

Reasons for using a label include the following;

1. A label makes a program location easier to find and remember.
2. The label can easily be moved to correct a program. The assembler will automatically change the address in all instructions that use the label when the program is reassembled.
3. The programmer does not have to calculate relative or absolute memory addresses, but just uses labels as needed.

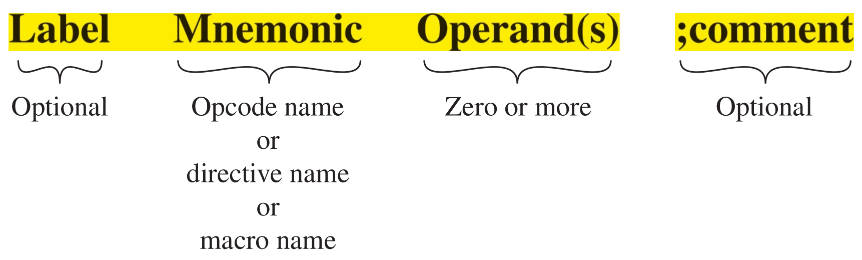

Figure B.1 Assembly-Language Statement Structure

MNEMONIC The mnemonic is the name of the operation or function of the assembly language statement. As discussed subsequently, a statement can correspond to a machine instruction, an assembler directive, or a macro. In the case of a machine instruction, a mnemonic is the symbolic name associated with a particular opcode.

Table 12.8 lists the mnemonic, or instruction name, of many of the x86 instructions. Appendix A of [CART06] lists the x86 instructions, together with the operands for each and the effect of the instruction on the condition codes. Appendix B of the NASM manual provides a more detailed description of each x86 instruction. Both documents are available at this book's Premium Content site.

OPERAND(S) An assembly language statement includes zero or more operands. Each operand identifies an immediate value, a register value, or a memory location. Typically, the assembly language provides conventions for distinguishing among the three types of operand references, as well as conventions for indicating addressing mode.

For the x86 architecture, an assembly language statement may refer to a register operand by name. Figure B.2 illustrates the general-purpose x86 registers, with their symbolic name and their bit encoding. The assembler will translate the symbolic name into the binary identifier for the register.

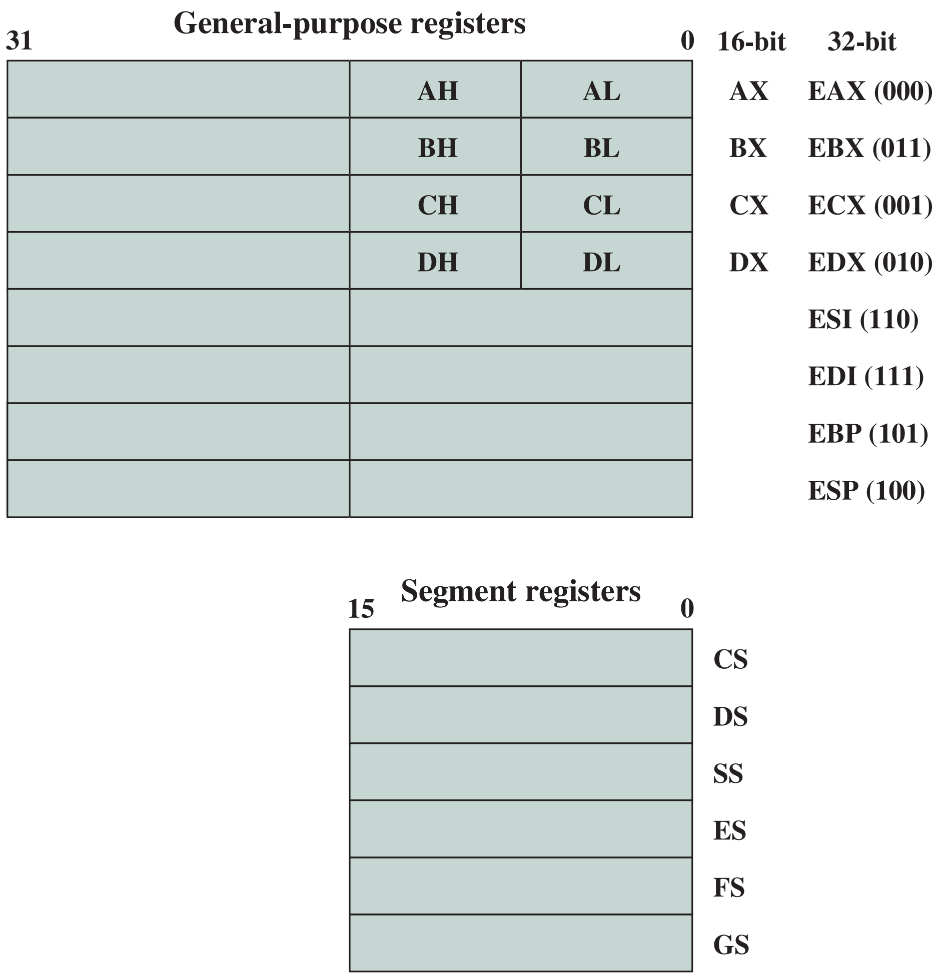

Figure B.2 Intel x86 Program Execution Registers

As discussed in Section 11.2, the x86 architecture has a rich set of addressing modes, each of which must be expressed symbolically in the assembly language. Here we cite a few of the common examples. For **register addressing**, the name of the register is used in the instruction. For example, `MOV ECX, EBX` copies the contents of register EBX into register ECX. Immediate addressing indicates that the value is encoded in the instruction. For example, `MOV EAX, 100H` copies the hexadecimal value 100 into register EAX. The immediate value can be expressed as a binary number with the suffix B or a decimal number with no suffix. Thus, equivalent statements to the preceding one are `MOV EAX, 100000000B` and `MOV EAX, 256`. **Direct addressing** refers to a memory location and is expressed as a displacement from the DS segment register. This is best explained by example. Assume that the 16-bit data segment register DS contains the value 1000H. Then the following sequence occurs:

```
MOV AX, 1234H
MOV [3518H], AX
```

First the 16-bit register AX is initialized to 1234H. Then, in line two, the contents of AX are moved to the logical address DS:3518H. This address is formed by shifting the contents of DS left 4 bits and adding 3518H to form the 32-bit logical address 13518H.

COMMENT All assembly languages allow the placement of comments in the program. A comment can either occur at the right-hand end of an assembly statement or can occupy an entire text line. In either case, the comment begins with a special character that signals to the assembler that the rest of the line is a comment and is to be ignored by the assembler. Typically, assembly languages for the x86 architecture use a semicolon (;) for the special character.

Type of Assembly Language Statements

Assembly language statements are one of four types: instruction, directive, macro definition, and comment. A comment statement is simply a statement that consists entirely of a comment. The remaining types are briefly described in this section.

INSTRUCTIONS The bulk of the noncomment statements in an assembly language program are symbolic representations of machine language instructions. Almost invariably, there is a one-to-one relationship between an assembly language instruction and a machine instruction. The assembler resolves any symbolic references and translates the assembly language instruction into the binary string that comprises the machine instruction.

DIRECTIVES Directives, also called **pseudo-instructions**, are assembly language statements that are not directly translated into machine language instructions. Instead, directives are instruction to the assembler to perform specified actions doing the assembly process. Examples include the following:

- Define constants
- Designate areas of memory for data storage
- Initialize areas of memory
- Place tables or other fixed data in memory
- Allow references to other programs

Table B.2 lists some of the NASM directives. As an example, consider the following sequence of statements:

Table B.2 Some NASM Assembly-Language Directives

(a) Letters for RES*x* and D*x* Directives

Unit	Letter
byte	B
word (2 bytes)	W
double word (4 bytes)	D
quad word (8 bytes)	Q
ten bytes	T

(b) Directives

Name	Description	Example

DB, DW, DD, DQ, DT	Initialize locations	`L6 DD 1A92H`
		`;doubleword at L6 initialized to 1A92H`

RESB, RESW, RESD, RESQ, REST	Reserve uninitialized locations	`BUFFER RESB 64`
		`;reserve 64 bytes starting at BUFFER`

INCBIN	Include binary file in output	`INCBIN "file.dat" ; include this file`

EQU	Define a symbol to a given constant value	`MSGLEN EQU 25`
		`;the constant MSGLEN equals decimal 25`

TIMES	Repeat instruction multiple times	`ZEROBUF TIMES 64 DB 0`
		`;initialize 64-byte buffer to all zeros`

```
L2 DB   "A"       ;byte initialized to ASCII code for A (65)
   MOV  AL, [L1]  ;copy byte at L1 into AL
   MOV  EAX, L1   ;store address of byte at L1 in EAX
   MOV  [L1], AH  ;copy contents of AH into byte at L1
```

If a plain label is used, it is interpreted as the address (or offset) of the data. If the label is placed inside square brackets, it is interpreted as the data at the address.

MACRO DEFINITIONS A macro definition is similar to a subroutine in several ways. A subroutine is a section of a program that is written once, and can be used multiple times by calling the subroutine from any point in the program. When a program is compiled or assembled, the subroutine is loaded only once. A call to the subroutine transfers control to the subroutine and a return instruction in the subroutine returns control to the point of the call. Similarly, a macro definition is a section of code that the programmer writes once, and then can use many times. The main difference is that when the assembler encounters a macro call, it replaces the macro call with the macro itself. This process is called **macro expansion**. So, if a macro is defined in an assembly language program and invoked 10 times, then 10 instances of the macro will appear in the assembled code. In essence, subroutines are handled by the hardware at run time, whereas macros are handled by the assembler at assembly time. Macros provide the same advantage as subroutines in terms of modular programming, but without the runtime overhead of a subroutine call and return. The tradeoff is that the macro approach uses more space in the object code.

In NASM and many other assemblers, a distinction is made between a single-line macro and a multi-line macro. In NASM, single-line macros are defined using the %DEFINE directive. Here is an example in which multiple single-line macros are expanded. First, we define two macros:

```
%DEFINE B(X) = 2*X
%DEFINE A(X) = 1 + B(X)
```

At some point in the assembly language program, the following statement appears:

```
MOV AX, A(8)
```

The assembler expands this statement to:

```
MOV AX, 1+2*8
```

which assembles to a machine instruction to move the immediate value 17 to register AX.

Multiline macros are defined using the mnemonic %MACRO. Here is an example of a multiline macro definition:

```
%MACRO PROLOGUE 1
       PUSH EBP      ;push contents of EBP onto stack
                     ;pointed to by ESP and
                     ;decrement contents of ESP by 4
       MOV EBP, ESP  ;copy contents of ESP to EBP
       SUB ESP, %1   ;subtract first parameter value from ESP
```

The number 1 after the macro name in the %MACRO line defines the number of parameters the macro expects to receive. The use of %1 inside the macro definition refers to the first parameter to the macro call.

The macro call

```
MYFUNC: PROLOGUE 12
```

expands to the following lines of code:

```
MYFUNC: PUSH  EBP
        MOV   EBP, ESP
        SUB   ESP, 12
```

Example: Greatest Common Divisor Program

As an example of the use of assembly language, we look at a program to compute the greatest common divisor of two integers. We define the greatest common divisor of the integers *a* and *b* as follows:

$$\gcd(a,b) = \max[k, \text{such that } k \text{ divides } a \text{ and } k \text{ divides } b]$$

where we say that k divides a if there is no remainder. Euclid's algorithm for the greatest common divisor is based on the following theorem. For any nonnegative integers a and b,

$$\gcd(a,b) = \gcd(b, a \bmod b)$$

Here is a C language program that implements Euclid's algorithm:

```
unsigned int gcd (unsigned int a, unsigned int b)

{
      if (a == 0 && b == 0)
          b = 1;
      else if (b == 0)
          b = a;
      else if (a != 0)
          while (a != b)
             if (a < b)
                 b -= a;
             else
                 a -= b;
      return b;
}
```

Figure B.3 shows two assembly language versions of the preceding program. The program on the left was done by a C compiler; the program on the right was programmed by hand. The latter program uses a number of programmer's tricks to produce a tighter, more efficient implementation.

B.2 ASSEMBLERS

The **assembler** is a software utility that takes an assembly program as input and produces object code as output. The object code is a binary file. The assembler views this file as a block of memory starting at relative location 0.

There are two general approaches to assemblers: the two-pass assembler and the one-pass assembler.

Two-Pass Assembler

We look first at the two-pass assembler, which is more common and somewhat easier to understand. The assembler makes two passes through the source code (Figure B.4):

FIRST PASS In the first pass, the assembler is only concerned with label definitions. The first pass is used to construct a **symbol table** that contains a list of all labels and their associated **location counter** (LC) values. The first byte of the object code will have the LC value of 0. The first pass examines each assembly statement. Although the assembler is not yet ready to translate instructions, it must examine

```
gcd:        mov     ebx,eax
            mov     eax,edx
            test    ebx,ebx
            jne     L1
            test    edx,edx
            jne     L1
            mov     eax,1
            ret
L1:         test    eax,eax
            jne     L2
            mov     eax,ebx
            ret
L2:         test    ebx,ebx
            je      L5
L3:         cmp     ebx,eax
            je      L5
            jae     L4
            sub     eax,ebx
            jmp     L3
L4:         sub     ebx,eax
            jmp     L3
L5:         ret
```

(a) Compiled program

```
gcd:        neg     eax
            je      L3
L1:         neg     eax
            xchg    eax,edx
L2:         sub     eax,edx
            jg      L2
            jne     L1
L3:         add     eax,edx
            jne     L4
            inc     eax
L4:         ret
```

(b) Written directly in assembly language

Figure B.3 Assembly Programs for Greatest Common Divisor

each instruction sufficiently to determine the length of the corresponding machine instruction and therefore how much to increment the LC. This may require not only examining the opcode but also looking at the operands and the addressing modes.

Directives such as DQ and REST (see Table B.2) cause the location counter to be adjusted according to how much storage is specified.

When assembler encounters a statement with a label, it places the label into the symbol table, along with the current LC value. The assembler continues until it has read all of the assembly language statements.

SECOND PASS The second pass reads the program again from the beginning. Each instruction is translated into the appropriate binary machine code. Translation includes the following operations:

1. Translate the mnemonic into a binary opcode.
2. Use the opcode to determine the format of the instruction and the location and length of the various fields in the instruction.
3. Translate each operand name into the appropriate register or memory code.
4. Translate each immediate value into a binary string.
5. Translate any references to labels into the appropriate LC value using the symbol table.
6. Set any other bits in the instruction that are needed, including addressing mode indicators, condition code bits, and so on.

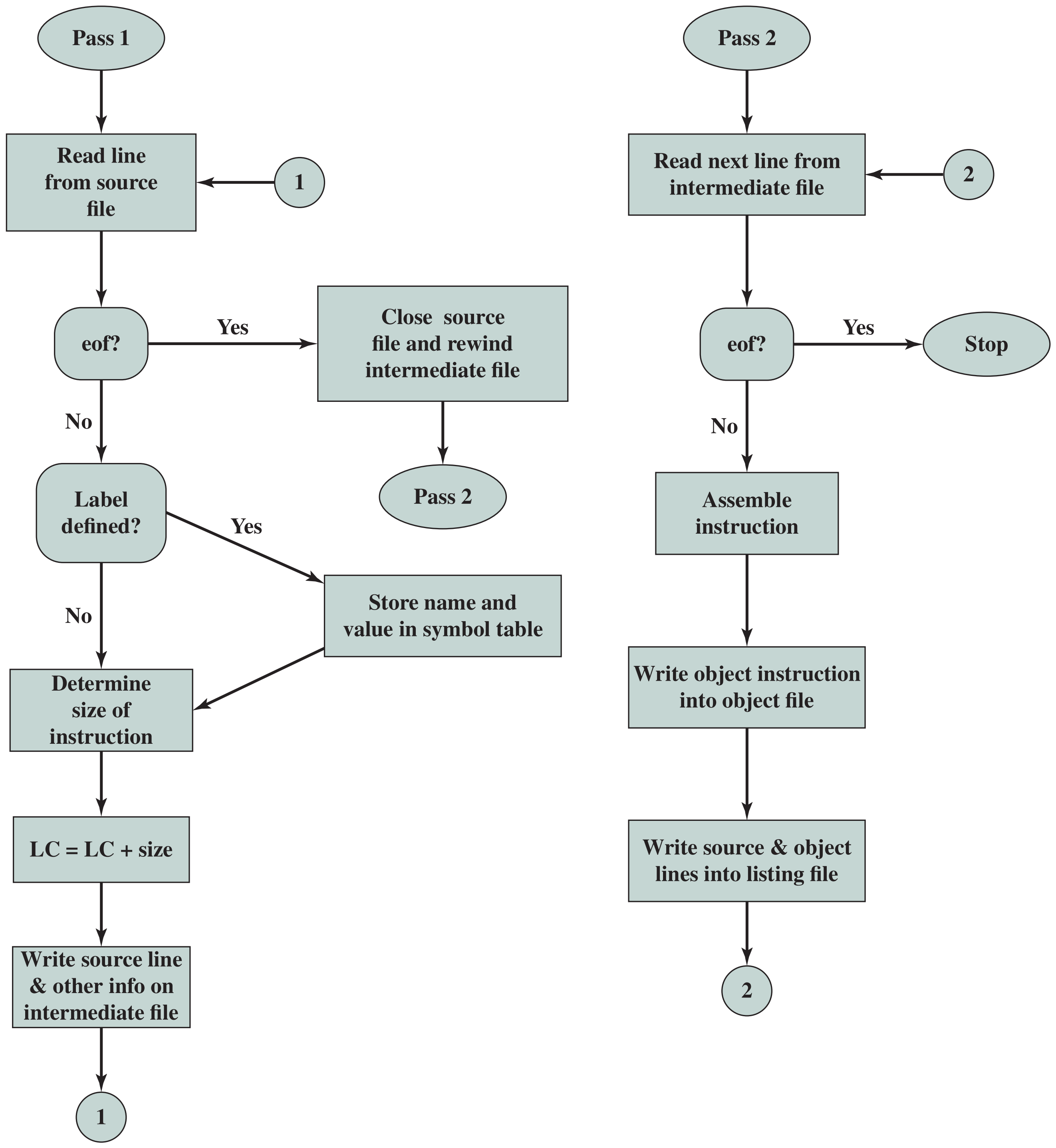
Pass 1
Read line from source file
1
eof?
Yes
Close source file and rewind intermediate file
Pass 2
No
Label defined?
Yes
Store name and value in symbol table
No
Determine size of instruction
LC = LC + size
Write source line & other info on intermediate file
1
Pass 2
Read next line from intermediate file
2
eof?
Yes
Stop
No
Assemble instruction
Write object instruction into object file
Write source & object lines into listing file
2

Figure B.4 Flowchart of Two-Pass Assembler

A simple example, using the ARM assembly language, is shown in Figure B.5. The ARM assembly language instruction ADDS r3, r3, #19 is translated in to the binary machine instruction 1110 0010 0101 0011 0011 0000 0001 0011.

ZEROTH PASS Most assembly language includes the ability to define macros. When macros are present there is an additional pass that the assembler must make before the first pass. Typically, the assembly language requires that all macro definitions must appear at the beginning of the program.

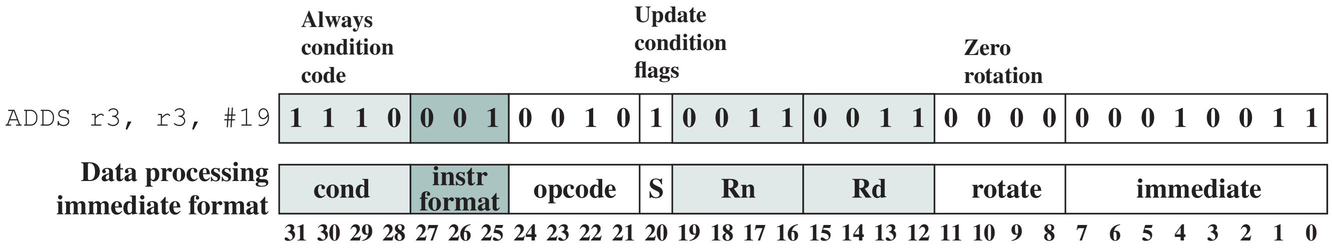

Figure B.5 Translating an ARM Assembly Instruction into a Binary Machine Instruction

The assembler begins this "zeroth pass" by reading all macro definitions. Once all the macros are recognized, the assembler goes through the source code and expands the macros with their associated parameters whenever a macro call is encountered. The macro processing pass generates a new version of the source code with all of the macro expansions in place and all of the macro definitions removed.

One-Pass Assembler

It is possible to implement an assembler that makes only a single pass through the source code (not counting the macro processing pass). The main difficulty in trying to assemble a program in one pass involves forward references to labels. Instruction operands may be symbols that have not yet been defined in the source program. Therefore, the assembler does not know what relative address to insert in the translated instruction.

In essence, the process of resolving forward references works as follows. When the assembler encounters an instruction operand that is a symbol that is not yet defined, the assembler does the following:

1. It leaves the instruction operand field empty (all zeros) in the assembled binary instruction.
2. The symbol used as an operand is entered in the symbol table. The table entry is flagged to indicate that the symbol is undefined.
3. The address of the operand field in the instruction that refers to the undefined symbol is added to a list of forward references associated with the symbol table entry.

When the symbol definition is encountered so that a LC value can be associated with it, the assembler inserts the LC value in the appropriate entry in the symbol table. If there is a forward reference list associated with the symbol, then the assembler inserts the proper address into any instruction previously generated that is on the forward reference list.

Example: Prime Number Program

We now look at an example that includes directives. This example looks at a program that finds prime numbers. Recall that prime numbers are evenly divisible by only 1 and themselves. There is no formula for doing this. The basic method this program uses is to find the factors of all odd numbers below a given limit. If no factor can be

```
unsigned guess;                               /* current guess for prime */
unsigned factor;                              /* possible factor of guess */
unsigned limit;                               /* find primes up to this value */

printf ("Find primes up to : ");
scanf("%u", &limit);
printf ("2\n");                               /* treat first two primes as */
printf ("3\n");                               /* special case */
guess = 5;                                    /* initial guess */
while (guess < = limit) {                     /* look for a factor of guess */
    factor = 3;
    while (factor * factor < guess && guess% factor != 0)
    factor + = 2;
    if (guess % factor != 0)
        printf ("%d\n", guess);
    guess += 2;                               /* only look at odd numbers */
}
```

Figure B.6 C Program for Testing Primality

found for an odd number, it is prime. Figure B.6 shows the basic algorithm written in C. Figure B.7 shows the same algorithm written in NASM assembly language.

B.3 LOADING AND LINKING

The first step in the creation of an active process is to load a program into main memory and create a process image (Figure B.8). Figure B.9 depicts a scenario typical for most systems. The application consists of a number of compiled or assembled modules in object-code form. These are linked to resolve any references between modules. At the same time, references to library routines are resolved. The library routines themselves may be incorporated into the program or referenced as shared code that must be supplied by the operating system at run time. In this section, we summarize the key features of linkers and loaders. First, we discuss the concept of relocation. Then, for clarity in the presentation, we describe the loading task when a single program module is involved; no linking is required. We can then look at the linking and loading functions as a whole.

Relocation

In a multiprogramming system, the available main memory is generally shared among a number of processes. Typically, it is not possible for the programmer to know in advance which other programs will be resident in main memory at the time of execution of his or her program. In addition, we would like to be able to swap active processes in and out of main memory to maximize processor utilization by providing a large pool of ready processes to execute. Once a program has been swapped out to disk, it would be quite limiting to declare that when it is next swapped back in, it must be placed in the same main memory region as before. Instead, we may need to **relocate** the process to a different area of memory.

```
%include "asm_io.inc"
segment .data
Message db "Find primes up to: ", 0

segment .bss
Limit resd 1                    ; find primes up to this limit
Guess resd 1                    ; the current guess for prime

segment .text
     global _asm_main
_asm_main:
     enter 0,0                  ; setup routine
     pusha

     mov eax, Message
     call print_string
     call read_int              ; scanf("%u", & limit);
     mov [Limit], eax
     mov eax, 2                 ; printf("2\n");
     call print_int
     call print_nl
     mov eax, 3                 ; printf("3\n");
     call print_int
     call print_nl

     mov dword [Guess], 5       ; Guess = 5;
while_limit:                    ; while (Guess <= Limit)
     mov eax, [Guess]
     cmp eax, [Limit]
     jnbe end_while_limit       ; use jnbe since numbers are unsigned

     mov ebx, 3                 ; ebx is factor = 3;
while_factor:
     mov eax,ebx
     mul eax                    ; edx:eax = eax*eax
     jo end_while_factor        ; if answer won't fit in eax alone
     cmp eax, [Guess]
     jnb end_while_factor       ; if !(factor*factor < guess)
     mov eax,[Guess]
     mov edx,0
     div ebx                    ; edx = edx:eax% ebx
     cmp edx, 0
     je end_while_factor        ; if !(guess% factor != 0)

     add ebx,2; factor += 2;
     jmp while_factor
end_while_factor:
     je end_if                  ; if !(guess% factor != 0)
     mov eax,[Guess]            ; printf("%u\n")
     call print_int
     call print_nl
end_if:
     add dword [Guess], 2       ; guess += 2
     jmp while_limit
end_while_limit:

     popa
     mov eax, 0                 ; return back to C
     leave
     ret
```

Figure B.7 Assembly Program for Testing Primality

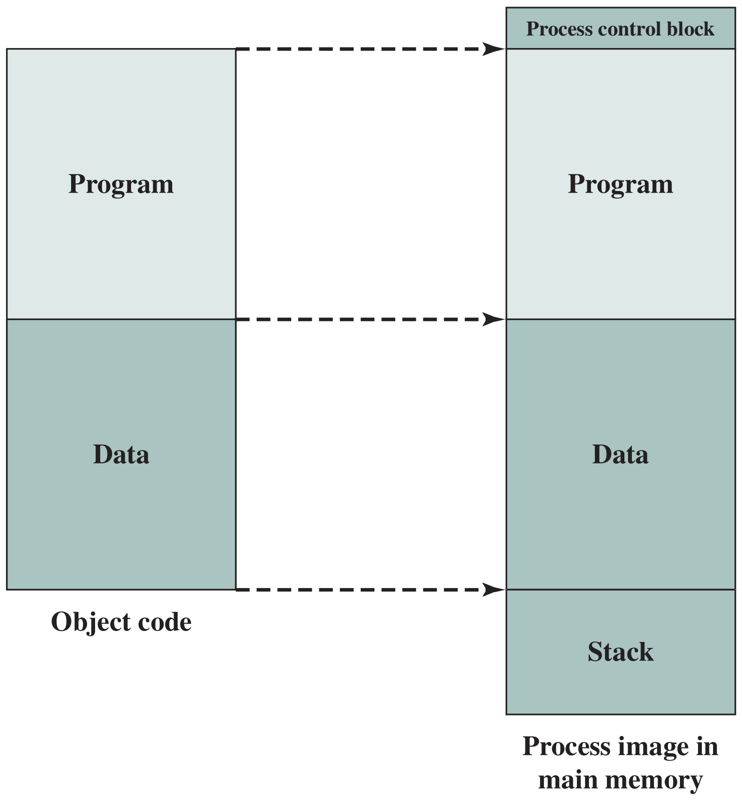

Figure B.8 The Loading Function

Thus, we cannot know ahead of time where a program will be placed, and we must allow that the program may be moved about in main memory due to swapping. These facts raise some technical concerns related to addressing, as illustrated in Figure B.10. The figure depicts a process image. For simplicity, let us assume that the process image occupies a contiguous region of main memory. Clearly, the

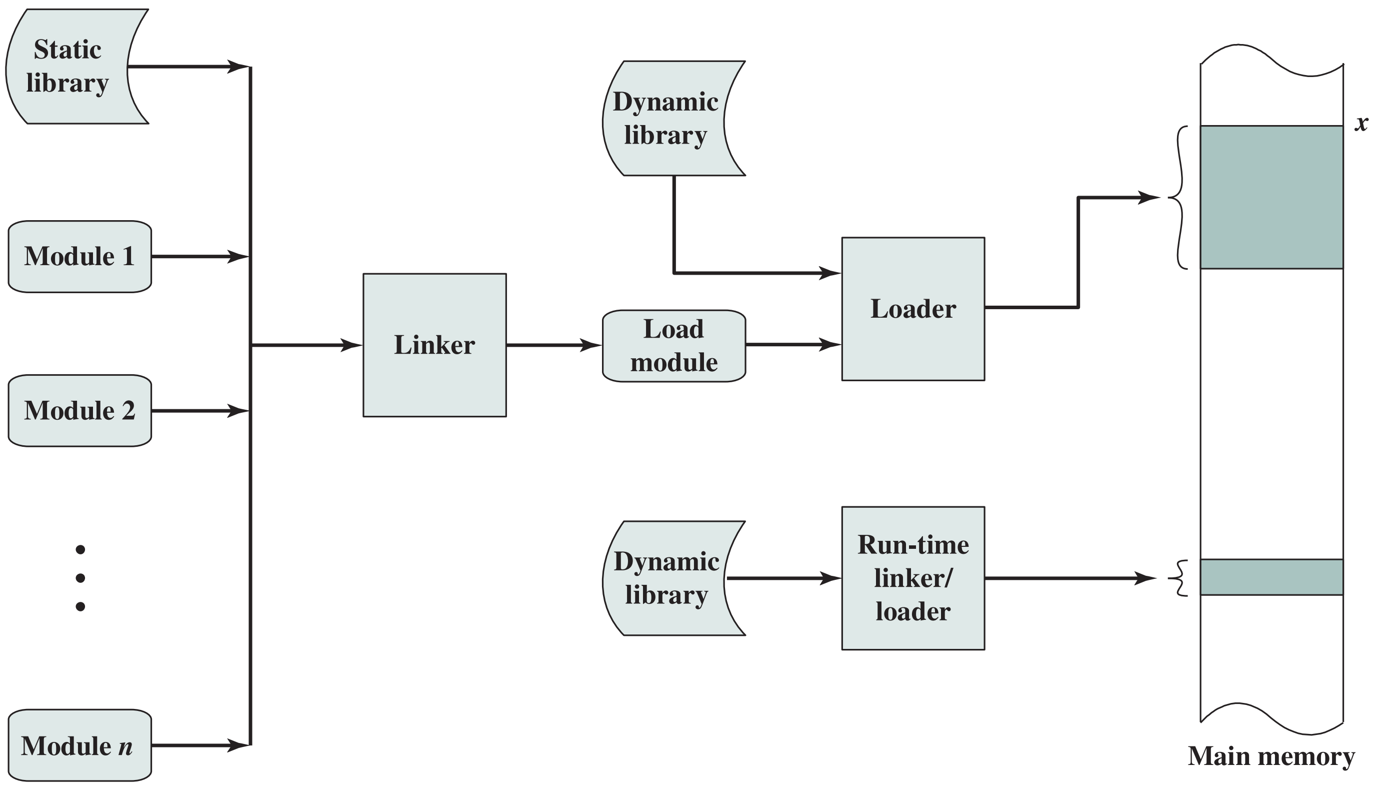

Figure B.9 A Linking and Loading Scenario

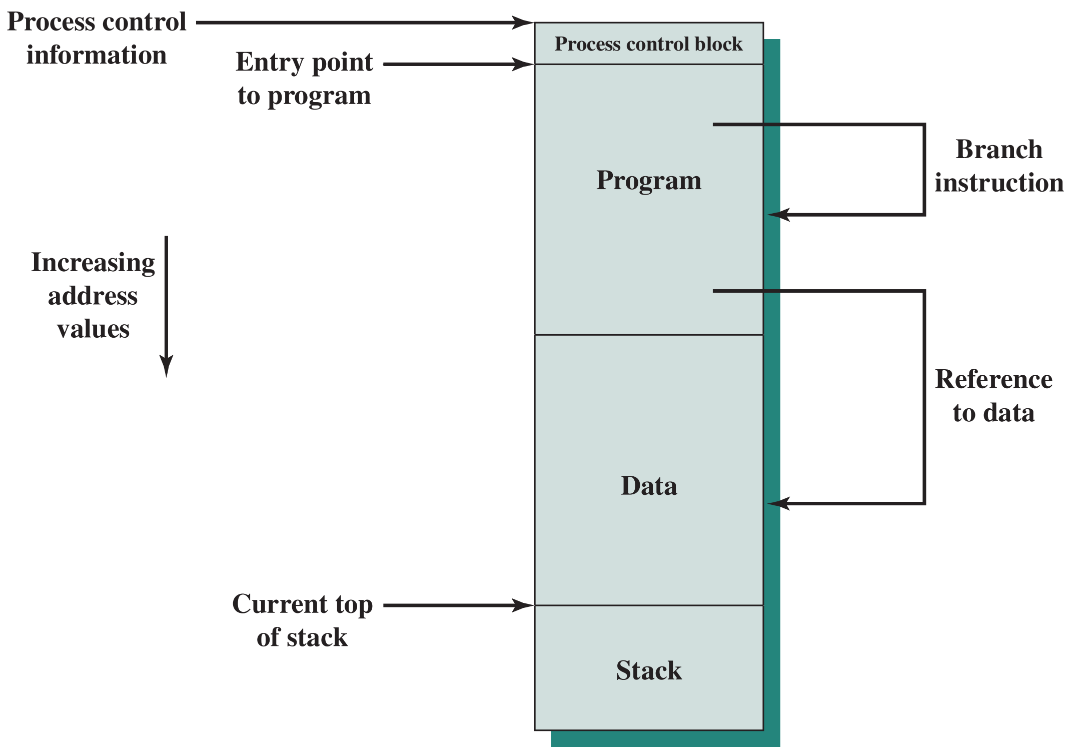

Figure B.10 Addressing Requirements for a Process

operating system will need to know the location of process control information and of the execution stack, as well as the entry point to begin execution of the program for this process. Because the operating system is managing memory and is responsible for bringing this process into main memory, these addresses are easy to come by. In addition, however, the processor must deal with memory references within the program. Branch instructions contain an address to reference the instruction to be executed next. Data reference instructions contain the address of the byte or word of data referenced. Somehow, the processor hardware and operating system software must be able to translate the memory references found in the code of the program into actual physical memory addresses, reflecting the current location of the program in main memory.

Loading

In Figure B.9, the loader places the load module in main memory starting at location *x*. In loading the program, the addressing requirement illustrated in Figure B.10 must be satisfied. In general, three approaches can be taken:

- Absolute loading
- Relocatable loading
- Dynamic run-time loading

ABSOLUTE LOADING An absolute loader requires that a given load module always be loaded into the same location in main memory. Thus, in the load module presented to the loader, all address references must be to specific, or absolute, main memory addresses. For example, if *x* in Figure B.9 is location 1024, then the first word in a load module destined for that region of memory has address 1024.

The assignment of specific address values to memory references within a program can be done either by the programmer or at compile or assembly time (Table B.3a). There are several disadvantages to the former approach. First, every programmer would have to know the intended assignment strategy for placing modules into main memory. Second, if any modifications are made to the program that involve insertions or deletions in the body of the module, then all of the addresses will have to be altered. Accordingly, it is preferable to allow memory references within programs to be expressed symbolically and then resolve those symbolic references at the time of compilation or assembly. This is illustrated in Figure B.11. Every reference to an instruction or item of data is initially represented by a symbol. In preparing the module for input to an absolute loader, the assembler or compiler will convert all of these references to specific addresses (in this example, for a module to be loaded starting at location 1024), as shown in Figure B.11b.

Table B.3 Address Binding

(a) Loader

Binding Time	Function
Programming time	All actual physical addresses are directly specified by the programmer in the program itself.
Compile or assembly time	The program contains symbolic address references, and these are converted to actual physical addresses by the compiler or assembler.
Load time	The compiler or assembler produces relative addresses. The loader translates these to absolute addresses at the time of program loading.
Run time	The loaded program retains relative addresses. These are converted dynamically to absolute addresses by processor hardware.

(b) Linker

Linkage Time	Function
Programming time	No external program or data references are allowed. The programmer must place into the program the source code for all subprograms that are referenced.
Compile or assembly time	The assembler must fetch the source code of every subroutine that is referenced and assemble them as a unit.
Load module creation	All object modules have been assembled using relative addresses. These modules are linked together and all references are restated relative to the origin of the final load module.
Load time	External references are not resolved until the load module is to be loaded into main memory. At that time, referenced dynamic link modules are appended to the load module, and the entire package is loaded into main or virtual memory.
Run time	External references are not resolved until the external call is executed by the processor. At that time, the process is interrupted and the desired module is linked to the calling program.

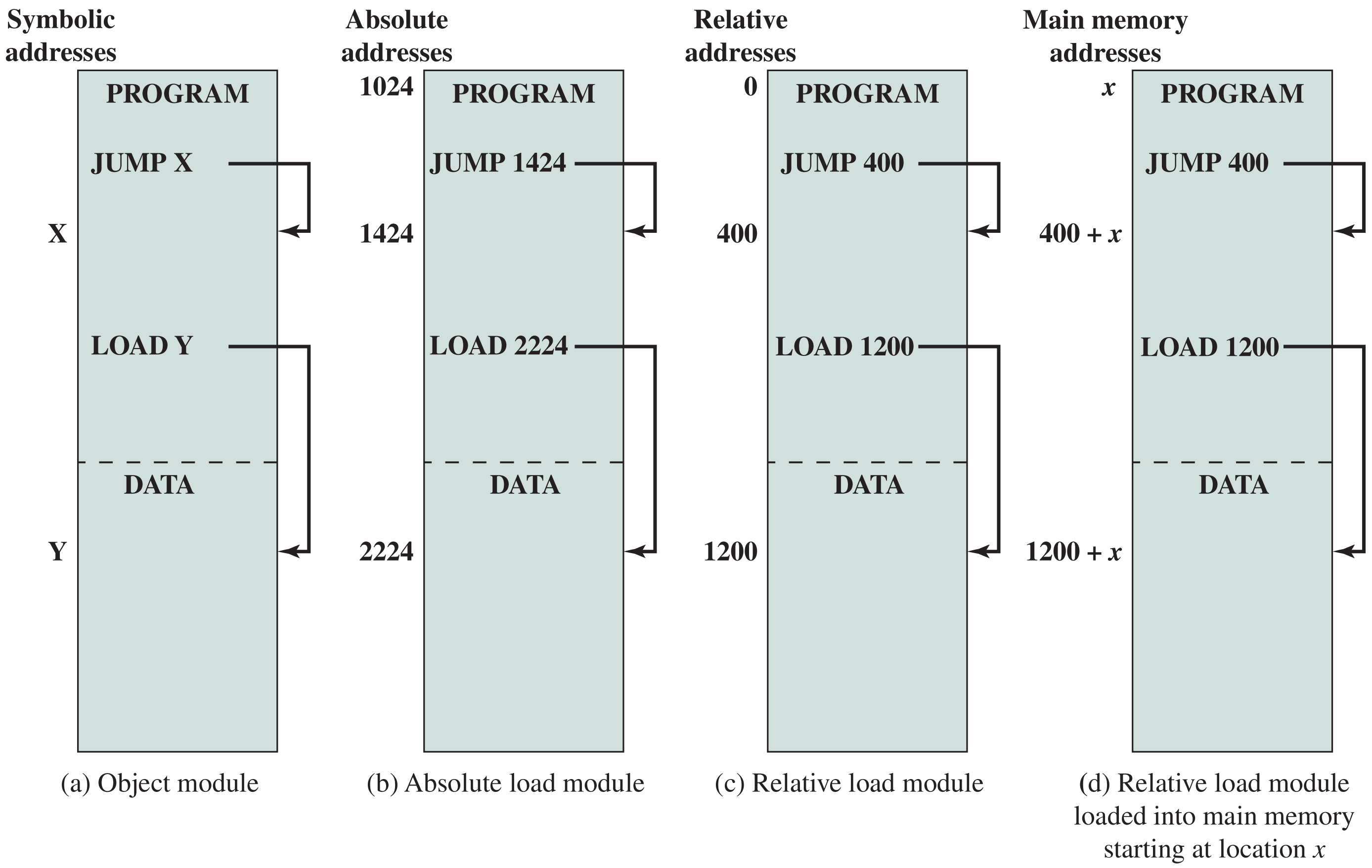

Figure B.11 Absolute and Relocatable Load Modules

RELOCATABLE LOADING The disadvantage of binding memory references to specific addresses prior to loading is that the resulting load module can only be placed in one region of main memory. However, when many programs share main memory, it may not be desirable to decide ahead of time into which region of memory a particular module should be loaded. It is better to make that decision at load time. Thus we need a load module that can be located anywhere in main memory.

To satisfy this new requirement, the assembler or compiler produces not actual main memory addresses (absolute addresses) but addresses that are relative to some known point, such as the start of the program. This technique is illustrated in Figure B.11c. The start of the load module is assigned the relative address 0, and all other memory references within the module are expressed relative to the beginning of the module.

With all memory references expressed in relative format, it becomes a simple task for the loader to place the module in the desired location. If the module is to be loaded beginning at location x, then the loader must simply add x to each memory reference as it loads the module into memory. To assist in this task, the load module must include information that tells the loader where the address references are and how they are to be interpreted (usually relative to the program origin, but also possibly relative to some other point in the program, such as the current location). This set of information is prepared by the compiler or assembler and is usually referred to as the relocation dictionary.

DYNAMIC RUN-TIME LOADING Relocatable loaders are common and provide obvious benefits relative to absolute loaders. However, in a multiprogramming environment, even one that does not depend on virtual memory, the relocatable loading scheme is inadequate. We have referred to the need to swap process images in and out of main memory to maximize the utilization of the processor. To maximize main memory utilization, we would like to be able to swap the process image back into different locations at different times. Thus, a program, once loaded, may be swapped out to disk and then swapped back in at a different location. This would be impossible if memory references had been bound to absolute addresses at the initial load time.

The alternative is to defer the calculation of an absolute address until it is actually needed at run time. For this purpose, the load module is loaded into main memory with all memory references in relative form (Figure B.11c). It is not until an instruction is actually executed that the absolute address is calculated. To assure that this function does not degrade performance, it must be done by special processor hardware rather than software. This hardware is described in Chapter 8.

Dynamic address calculation provides complete flexibility. A program can be loaded into any region of main memory. Subsequently, the execution of the program can be interrupted and the program can be swapped out of main memory, to be later swapped back in at a different location.

Linking

The function of a linker is to take as input a collection of object modules and produce a load module, consisting of an integrated set of program and data modules, to be passed to the loader. In each object module, there may be address references to locations in other modules. Each such reference can only be expressed symbolically in an unlinked object module. The linker creates a single load module that is the contiguous joining of all of the object modules. Each intramodule reference must be changed from a symbolic address to a reference to a location within the overall load module. For example, module A in Figure B.12a contains a procedure invocation of module B. When these modules are combined in the load module, this symbolic reference to module B is changed to a specific reference to the location of the entry point of B within the load module.

LINKAGE EDITOR The nature of this address linkage will depend on the type of load module to be created and when the linkage occurs (Table B.3b). If, as is usually the case, a relocatable load module is desired, then linkage is usually done in the following fashion. Each compiled or assembled object module is created with references relative to the beginning of the object module. All of these modules are put together into a single relocatable load module with all references relative to the origin of the load module. This module can be used as input for relocatable loading or dynamic run-time loading.

A linker that produces a relocatable load module is often referred to as a linkage editor. Figure B.12 illustrates the linkage editor function.

DYNAMIC LINKER As with loading, it is possible to defer some linkage functions. The term *dynamic linking* is used to refer to the practice of deferring the linkage of some external modules until after the load module has been created. Thus, the load module contains unresolved references to other programs. These references can be resolved either at load time or run time.

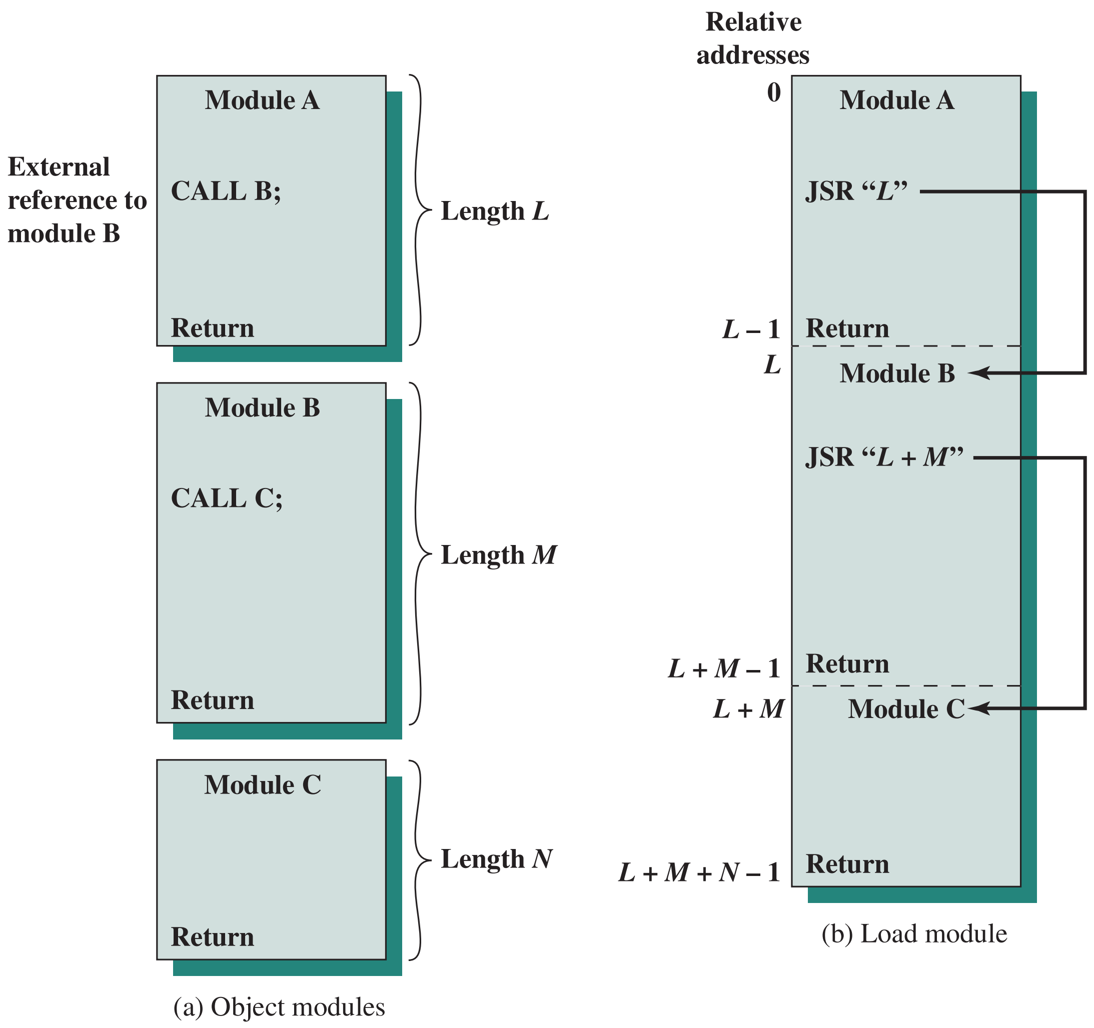

Figure B.12 The Linking Function

For **load-time dynamic linking** (involving upper dynamic library in Figure B.9), the following steps occur. The load module (application module) to be loaded is read into memory. Any reference to an external module (target module) causes the loader to find the target module, load it, and alter the reference to a relative address in memory from the beginning of the application module. There are several advantages to this approach over what might be called static linking:

- It becomes easier to incorporate changed or upgraded versions of the target module, which may be an operating system utility or some other general-purpose routine. With static linking, a change to such a supporting module would require the relinking of the entire application module. Not only is this inefficient, but it may be impossible in some circumstances. For example, in the personal computer field, most commercial software is released in load module form; source and object versions are not released.
- Having target code in a dynamic link file paves the way for automatic code sharing. The operating system can recognize that more than one application is using the same target code because it loaded and linked that code. It can use that information to load a single copy of the target code and link it to both applications, rather than having to load one copy for each application.

- It becomes easier for independent software developers to extend the functionality of a widely used operating system such as Linux. A developer can come up with a new function that may be useful to a variety of applications and package it as a dynamic link module.

With **run-time dynamic linking** (involving lower dynamic library in Figure B.9), some of the linking is postponed until execution time. External references to target modules remain in the loaded program. When a call is made to the absent module, the operating system locates the module, loads it, and links it to the calling module. Such modules are typically shareable. In the Windows environment, these are called dynamic-link libraries (DLLs) Thus, if one process is already making use of a dynamically linked shared module, then that module is in main memory and a new process can simply link to the already-loaded module.

The use of DLLs can lead to a problem commonly referred to as **DLL hell**. DLL hell occurs if two or more processes are sharing a DLL module but expect different versions of the module. For example, an application or system function might be re-installed and bring in with it an older version of a DLL file.

We have seen that dynamic loading allows an entire load module to be moved around; however, the structure of the module is static, being unchanged throughout the execution of the process and from one execution to the next. However, in some cases, it is not possible to determine prior to execution which object modules will be required. This situation is typified by transaction-processing applications, such as an airline reservation system or a banking application. The nature of the transaction dictates which program modules are required, and they are loaded as appropriate and linked with the main program. The advantage of the use of such a dynamic linker is that it is not necessary to allocate memory for program units unless those units are referenced. This capability is used in support of segmentation systems.

One additional refinement is possible: An application need not know the names of all the modules or entry points that may be called. For example, a charting program may be written to work with a variety of plotters, each of which is driven by a different driver package. The application can learn the name of the plotter that is currently installed on the system from another process or by looking it up in a configuration file. This allows the user of the application to install a new plotter that did not exist at the time the application was written.

B.4 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

Assembler	label	mnemonic
assembly language	linkage editor	one-pass assembler
comment	linking	operand
directive	load-time dynamic linking	relocation
dynamic linker	loading	run-time dynamic linking
instruction	macro	two-pass assembler

Review Questions

B.1 List some reasons why it is worthwhile to study assembly language programming.

B.2 What is an assembly language?

B.3 List some disadvantages of assembly language compared to high-level languages.

B.4 List some advantages of assembly language compared to high-level languages.

B.5 What are the typical elements of an assembly language statement?

B.6 List and briefly define four different kinds of assembly language statements.

B.7 What is the difference between a one-pass assembler and a two-pass assembler?

Problems

B.1 Core War is a programming game introduced to the public in the early 1980s [DEWD84], which was popular for a period of 15 years or so. Core War has four main components: a memory array of 8000 addresses; a simplified assembly language Redcode; an executive program called MARS (an acronym for Memory Array Redcode Simulator); and the set of contending battle programs. Two battle programs are entered into the memory array at randomly chosen positions; neither program knows where the other one is. MARS executes the programs in a simple version of time-sharing. The two programs take turns; a single instruction of the first program is executed, then a single instruction of the second, and so on. What a battle program does during the execution cycles allotted to it is entirely up to the programmer. The aim is to destroy the other program by ruining its instructions. In this problem and the next several, we use an even simpler language, called CodeBlue, to explore some Core War concepts.

CodeBlue contains only five assembly language statements and uses three addressing modes (Table B.4). Addresses wrap around, so that for the last location in memory, the relative address of +1 refers to the first location in memory. For example, `ADD #4,6` adds 4 to the contents of relative location 6 and stores the results in location 6; `JUMP @5` transfers execution to the memory address contained in the location five slots past the location of the current `JUMP` instruction.

a. The program Imp is the single instruction `COPY 0,1`. What does it do?

b. The program Dwarf is the following sequence of instructions:

```
ADD #4, 3
COPY 2, @2
JUMP -2
DATA 0
```

What does it do?

c. Rewrite Dwarf using symbols, so that it looks more like a typical assembly language program.

B.2 What happens if we pit Imp against Dwarf?

B.3 Write a “carpet bombing” program in CodeBlue that zeros out all of memory (with the possible exception of the program locations).

B.4 How would the previous program fare against Imp?

B.5 **a.** What is the value of the C status flag after the following sequence:

```
mov al, 3
add al, 4
```

b. What is the value of the C status flag after the following sequence:

```
mov al, 3
sub al, 4
```

Table B.4 CodeBlue Assembly Language

(a) Instruction Set

Format		Meaning
DATA	<value>	<value> set at current location
COPY	A, B	copies source A to destination B
ADD	A, B	adds A to B, putting result in B
JUMP	A	transfer execution to A
JUMPZ	A, B	if B = 0, transfer to A

(b) Addressing Modes

Mode	Format	Meaning
Literal	# followed by value	This is an immediate mode, the operand value is in the instruction.
Relative	Value	The value represents an offset from the current location, which contains the operand.
Indirect	@ followed by value	The value represents an offset from the current location; the offset location contains the relative address of the location that contains the operand.

```
Loop COPY #0, -1
     JUMP -1
```

Hint: Remember that instruction execution alternates between the two opposing programs.

B.6 Consider the following NAMS instruction:

```
cmp vleft, vright
```

For signed integers, there are three status flags that are relevant. If vleft = vright, then ZF is set. If vleft > vright, ZF is unset (set to 0) and SF = OF. If vleft < vright, ZF is unset and SF ≠ OF. Why does SF = OF if vleft > vright?

B.7 Consider the following NASM code fragment:

```
mov al, 0
cmp al, al
je next
```

Write an equivalent program consisting of a single instruction.

B.8 Consider the following C program:

```
/* a simple C program to average 3 integers */
main ()
{ int avg;
  int i1 = 20;
  int i2 = 13;
  int i3 = 82;
  avg = (i1 + i2 + i3)/3;
}
```

Write an NASM version of this program.

B.9 Consider the following C code fragment:

```
if (EAX == 0) EBX = 1;
else EBX = 2;
```

Write an equivalent NASM code fragment.

B.10 The initialize data directives can be used to initialize multiple locations. For example,

```
db 0x55,0x56,0x57
```

reserves three bytes and initializes their values.

NASM supports the special token $ to allow calculations to involve the current assembly position. That is, $ evaluates to the assembly position at the beginning of the line containing the expression. With the preceding two facts in mind, consider the following sequence of directives:

```
message db 'hello, world'
msglen equ $-message
```

What value is assigned to the symbol msglen?

B.11 Assume the three symbolic variables V1, V2, V3 contain integer values. Write an NASM code fragment that moves the smallest value into integer ax. Use only the instructions `mov`, `cmp`, and `jbe`.

B.12 Describe the effect of this instruction: `cmp eax,1` Assume that the immediately preceding instruction updated the contents of eax.

B.13 The `xchg` instruction can be used to exchange the contents of two registers. Suppose that the x86 instruction set did not support this instruction.

a. Implement `xchg ax,bx` using only push and pop instructions.

b. Implement `xchg ax, bx` using only the `xor` instruction (do not involve other registers).

B.14 In the following program, assume that a, b, x, y are symbols for main memory locations. What does the program do? You can answer the question by writing the equivalent logic in C.

```
       mov    eax,a
       mov    ebx,b
       xor    eax,x
       xor    ebx,y
       or     eax,ebx
       jnz    L2
L1:                  ;sequence of instructions…
       jmp    L3
L2:                  ;another sequence of instructions…
L3:
```

B.15 Section B.1 includes a C program that calculates the greatest common divisor of two integers.

a. Describe the algorithm in words and show how the program does implement the Euclid algorithm approach to calculating the greatest common divisor.

b. Add comments to the assembly program of Figure B.3a to clarify that it implements the same logic as the C program.

c. Repeat part (b) for the program of Figure B.3b.

B.16 **a.** A 2-pass assembler can handle future symbols and an instruction can therefore use a future symbol as an operand. This is not always true for directives. The EQU directive, for example, cannot use a future symbol. The directive "A EQU B + 1" is easy to execute if B is previously defined, but impossible if B is a future symbol. What's the reason for this?

b. Suggest a way for the assembler to eliminate this limitation such that any source line could use future symbols.

B.17 Consider a symbol directive MAX of the following form: symbol MAX list of expressions

The label is mandatory and is assigned the value of the largest expression in the operand field. Example:

```
MSGLEN MAX A, B, C ;where A, B, C are defined symbols
```

How is MAX executed by the Assembler and in what pass?