the addresses of these variables. Thus the second line in Fig. 5.14(c) specifies that one variable is to be read, and the third line gives the address of this variable. The address of the first word of the parameter list will automatically be placed in register L by the JSUB instruction. The subroutine XREAD can use this address to locate its parameters, and then add the length of the parameter list to register L to find the true return address.

Figure 5.14(b) shows a set of routines that might be used to accomplish this code generation. The first two routines correspond to alternative structures for <id-list>, which are shown in Rule 6 of the grammar in Fig. 5.2. In either case, the token specifier S(**id**) for a new identifier being added to the <id-list> is inserted into the list used by the code-generation routines, and LISTCOUNT is updated to reflect this insertion. After the entire <id-list> has been parsed, the list contains the token specifiers for all the identifiers that are part of the <id-list>. When the <read> statement is recognized, these token specifiers are removed from the list and used to generate the object code for the READ.

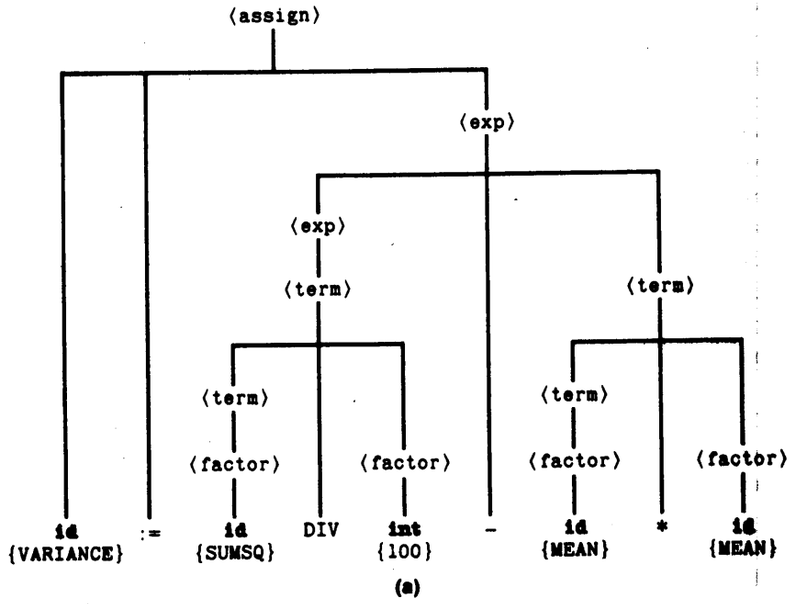
Remember that the parser, in generating the tree shown in Fig. 5.14(a), recognizes first <id-list> and then <read>. At each step, the parser calls the appropriate code-generation routine. You should work through this example carefully to be sure you understand how the code-generation routines in Fig. 5.14(b) create the object code that is symbolically represented in Fig. 5.14(c).

Figure 5.15 shows the code-generation process for the assignment statement on line 14 of Fig. 5.1. Figure 5.15(a) displays the parse tree for this statement. Most of the work of parsing involves the analysis of the <exp> on the right-hand side of the :=. As we can see, the parser first recognizes the **id** SUMSQ as a <factor> and a <term>; then it recognizes the **int** 100 as a <factor>; then it recognizes SUMSQ DIV 100 as a <term>, and so forth. The order in which the parts of the statement are recognized is the same as the order in which the calculations are to be performed: SUMSQ DIV 100 and MEAN * MEAN are computed, and then the second result is subtracted from the first.

As each portion of the statement is recognized, a code-generation routine is called to create the corresponding object code. For example, suppose we want to generate code that corresponds to the rule

$$\texttt{<term>}_1 \texttt{ ::= <term>}_2 \texttt{ * <factor>}$$

The subscripts are used here to distinguish between the two occurrences of <term>. Our code-generation routines perform all arithmetic operations using register A, so we clearly need to generate a MUL instruction in the object code. The result of this multiplication, <term>$_1$, will be left in register A by the MUL. If either <term>$_2$ or <factor> is already present in register A, perhaps as the result of a previous computation, the MUL instruction is all we need. Otherwise,



(a)

```
<assign> ::= id := <exp>

                GETA (<exp>)
                generate [STA S(id)]
                REGA := null


<exp> ::= <term>

                S(<exp>) := S(<term>)
                if S(<exp>) = rA then
                    REGA := <exp>


<exp>1 ::= <exp>2 + <term>

                if S(<exp>2) = rA then
                    generate [ADD S(<term>)]
                else if S(<term>) = rA then
                    generate [ADD S(<exp>2)]
                else
                    begin
                        GETA (<exp>2)
                        generate [ADD S(<term>)]
                    end
                S(<exp>1) := rA
                REGA := <exp>1
```

Figure 5.15 Code generation for an assignment statement.

```
<exp>1 ::= <exp>2 - <term>

            if S(<exp>2) = rA then
               generate [SUB S(<term>)]
            else
               begin
                  GETA (<exp>2)
                  generate [SUB S(<term>)]
               end
            S(<exp>1) := rA
            REGA := <exp>1


<term> ::= <factor>

            S(<term>) := S(<factor>)
            if S(<term>) = rA then
               REGA := <term>


<term>1 ::= <term>2 * <factor>

            if S(<term>2) = rA then
               generate [MUL S(<factor>)]
            else if S(<factor>) = rA then
               generate [MUL S(<term>2)]
            else
               begin
                  GETA (<term>2)
                  generate [MUL S(<factor>)]
               end
            S(<term>1) := rA
            REGA := <term>1


<term>1 ::= <term>2 DIV <factor>

            if S(<term>2) = rA then
               generate [DIV S(<factor>)]
            else
               begin
                  GETA (<term>2)
                  generate [DIV S(<factor>)]
               end
            S(<term>1) := rA
            REGA := <term>1
```

Figure 5.15 (*cont'd*)

```
<factor> ::= id
             S(<factor>) := S(id)

<factor> ::= int
             S(<factor>) := S(int)

<factor> ::= (<exp>)
             S(<factor>) := S(<exp>)
             if S(<factor>) = rA then
                REGA := <factor>
```

(b)

```
procedure GETA (NODE)
   begin
      if REGA = null then
         generate [LDA S(NODE)]
      else if S(NODE) π rA then
         begin
            create a new working variable Ti
            generate [STA Ti]
            record forward reference to Ti
            S(REGA) := Ti
            generate [LDA S(NODE)]
         end {if ≠ rA}
      S(NODE) := rA
      REGA := NODE
   end {GETA}
```

(c)

```
LDA     SUMSQ
DTV     #100
STA     T1
LDA     MEAN
MUL     MEAN
STA     T2
LDA     T1
SUB     T2
STA     VARIANCE
```

(d)

Figure 5.15 (*cont'd*)

we must generate a LDA instruction preceding the MUL. In that case we must also save the previous value in register A if it will be required for later use.

Obviously we need to keep track of the result left in register A by each segment of code that is generated. We do this by extending the token-specifier idea to nonterminal nodes of the parse tree. In the example just discussed, the *node specifier* S(<term>$_1$) would be set to rA, indicating that the result of this computation is in register A. The variable REGA is used to indicate the highest-level node of the parse tree whose value is left in register A by the code generated so far (i.e., the node whose specifier is rA). Clearly, there can be only one such node at any point in the code-generation process. If the value corresponding to a node is not in register A, the specifier for the node is similar to a token specifier: either a pointer to a symbol table entry for the variable that contains the value, or an integer constant.

As an illustration of these ideas, consider the code-generation routine in Fig. 5.15(b) that corresponds to the rule

```
<term>1 ::= <term>2 * <factor>
```

If the node specifier for either operand is rA, the corresponding value is already in register A, so the routine simply generates a MUL instruction. The operand address for this MUL is given by the node specifier for the other operand (the one not in the register). Otherwise, the procedure GETA is called. This procedure, shown in Fig. 5.15(c), generates a LDA instruction to load the value associated with <term>$_2$ into register A. Before the LDA, however, the procedure generates a STA instruction to save the value currently in register A (unless REGA is null, indicating that this value is no longer needed). The value is stored in a *temporary* variable. Such variables are created by the code generator (with names T1, T2,...) as they are needed for this purpose. The temporary variables used during a compilation will be assigned storage locations at the end of the object program. The node specifier for the node associated with the value previously in register A, indicated by REGA, is reset to indicate the temporary variable used.

After the necessary instructions are generated, the code-generation routine sets S(<term>$_1$) and REGA to indicate that the value corresponding to <term>$_1$ is now in register A. This completes the code-generation actions for the * operation.

The code-generation routine that corresponds to the + operation is almost identical to the one just discussed for *. The routines for DIV and – are similar, except that for these operations it is necessary for the first operand to be in register A. The code generation for <assign> consists of bringing the value to be assigned into register A (using GETA) and then generating a STA instruction. Note that REGA is then set to null because the code for the statement has been completely generated, and any intermediate results are no longer needed.

The remaining rules shown in Fig. 5.15(b) do not require the generation of any machine instructions since no computation or data movement is involved. The code-generation routines for these rules simply set the node specifier of the higher-level node to reflect the location of the corresponding value.

Figure 5.15(d) shows a symbolic representation of the object code generated for the assignment statement being translated. You should carefully work through the generation of this code to understand the operation of the routines in Fig. 5.15(b) and (c). You should also confirm that this code will perform the computations specified by the source program statement.

Figure 5.16 shows the other code-generation routines for the grammar in Fig. 5.2. The routine for <prog-name> generates header information in the object program that is similar to that created from the START and EXTREF assembler directives. It also generates instructions to save the return address and jump to the first executable instruction in the compiled program. When the complete <prog> is recognized, storage locations are assigned to any temporary (Ti) variables that have been used. Any references to these variables are then fixed in the object code using the same process performed for forward references by a one-pass assembler. The compiler also generates any Modification records required to describe external references to library subroutines.

Code generation for a <for> statement involves a number of steps. When the <index-exp> is recognized, code is generated to initialize the index variable for the loop and test for loop termination. Information is also saved on the stack for later use. Code is then generated separately for each statement in the body of the loop. When the complete <for> statement has been parsed, code is generated to increment the value of the index variable and to jump back to the beginning of the loop to test for termination. This code-generation routine uses the information saved on the stack by the routine for <index-exp>. Using a stack to store this information allows for nested <for> loops.

You are encouraged to trace through the complete code-generation process for this program, using the parse tree in Fig. 5.4 as a guide. The result should be as shown in Fig. 5.17. Once again, it is important to remember that this is merely a symbolic representation of the code generated. Most compilers would produce machine language code directly.

5.2 MACHINE-DEPENDENT COMPILER FEATURES

In this section we briefly discuss some machine-dependent extensions to the basic compilation scheme presented in Section 5.1. The purpose of a compiler is to translate programs written in a high-level programming language into the machine language for some computer. Most high-level programming languages are designed to be relatively independent of the machine being used

```
<prog> ::= PROGRAM <prog-name> VAR <dec-list> BEGIN <stmt-list> END.

            generate [LDL  RETADR]
            generate [RSUB]
            for each Ti variable used do
               generate [Ti RESW 1]
            insert [J EXADDR] {jump to first executable instruction}
               in bytes 3-5 of object program
            fix up forward references to Ti variables
            generate Modification records for external references
            generate [END]


<prog-name> ::= id

            generate [START 0]
            generate [EXTREF XREAD,XWRITE]
            generate [STL RETADR]
            add 3 to LOCCTR {leave room for jump to first executable instruction}
            generate [RETADR RESW 1]


<dec-list> ::= {either alternative}

            save LOCCTR as EXADDR {tentative address of first executable instruction}


<dec> ::= <id-list> : <type>

            for each item on list do
               begin
                  remove S(NAME) from list
                  enter LOCCTR into symbol table as address for NAME
                  generate [S(NAME)   RESW   1]
               end
            LISTCOUNT := 0


<type> ::= INTEGER

            {no code-generation action}


<stmt-list> ::= {either alternative}

            {no code-generation action}
```

Figure 5.16 Other code-generation routines for the grammar from Fig. 5.2.

```
<stmt> ::= {any alternative}

                {no code-generation action}

<write> ::= WRITE (<id-list>)

                generate [+JSUB XWRITE]
                record external reference to XWRITE
                generate [WORD LISTCOUNT]
                for each item on list do
                    begin
                        remove S(ITEM) from list
                        generate [WORDS (ITEM)]
                    end
                LISTCOUNT := 0

<for> ::= FOR <index-exp> DO <body>

                pop JUMPADDR from stack {address of jump out of loop}
                pop S(INDEX) from stack {index variable}
                pop LOOPADDR from stack {beginning address of loop}
                generate [LDA S(INDEX)]
                generate [ADD #1]
                generate [J LOOPADDR]
                insert [JGT LOCCTR] at location JUMPADDR

<index-exp> ::= id := <exp>1 TO <exp>2

                GETA (<exp>1)
                push LOCCTR onto stack {beginning address of loop}
                push S(id) onto stack {index variable}
                generate [STA S(id)]
                generate [COMP S(<exp>2)]
                push LOCCTR onto stack {address of jump out of loop}
                add 3 to LOCCTR {leave room for jump instruction}
                REGA := null

<body> ::= {either alternative}

                {no code-generation action}
```

Figure 5.16 (*cont'd*)

(although the extent to which this goal is realized varies considerably). This means that the process of analyzing the syntax of programs written in these languages should also be relatively machine-independent. It should come as no surprise, therefore, that the real machine dependencies of a compiler are related to the generation and optimization of the object code.

Line		Symbolic Representation of Generated Code		
1	STATS	START	0	{program header}
		EXTREF	XREAD,XWRITE	
		STL	RETADR	{save return address}
		J	{EXADDR}	
	RETADR	RESW	1	
3	SUM	RESW	1	{variable declarations}
	SUMSQ	RESW	1	
	I	RESW	1	
	VALUE	RESW	1	
	MEAN	RESW	1	
	VARIANCE	RESW	1	
5	{EXADDR}	LDA	#0	{SUM := 0}
		STA	SUM	
6		LDA	#0	{SUMSQ := 0}
		STA	SUMSQ	
7		LDA	#1	{FOR I := 1 TO 100}
	{L1}	STA	I	
		COMP	#100	
		JGT	{L2}	
9		+JSUB	XREAD	{READ(VALUE)}
		WORD	1	
		WORD	VALUE	
10		LDA	SUM	{SUM := SUM + VALUE}
		ADD	VALUE	
		STA	SUM	
11		LDA	VALUE	{SUMSQ := SUMSQ + VALUE * VALUE}
		MUL	VALUE	
		ADD	SUMSQ	
		STA	SUMSQ	
		LDA	I	{end of FOR loop}
		ADD	#1	
		J	{L1}	
13	{L2}	LDA	SUM	{MEAN := SUM DIV 100}
		DIV	#100	
		STA	MEAN	
14		LDA	SUMSQ	{VARIANCE := SUMSQ DIV 100 - MEAN * MEAN}
		DIV	#100	
		STA	T1	
		LDA	MEAN	
		MUL	MEAN	
		STA	T2	
		LDA	T1	
		SUB	T2	
		STA	VARIANCE	
15		+JSUB	XWRITE	{WRITE(MEAN,VARIANCE)}
		WORD	2	
		WORD	MEAN	
		WORD	VARIANCE	
		LDL	RETADR	{return}
		RSUB		
	T1	RESW	1	{working variables used}
	T2	RESW	1	
		END		

Figure 5.17 Symbolic representation of object code generated for the program from Fig. 5.1.

At an elementary level, of course, all code generation is machine-dependent because we must know the instruction set of a computer to generate code for it. However, there are many more complex issues involving such problems as the allocation of registers and the rearrangement of machine instructions to improve efficiency of execution. Such types of code optimization are normally done by considering an *intermediate form* of the program being compiled. In this intermediate form, the syntax and semantics of the source statements have been completely analyzed, but the actual translation into machine code has not yet been performed. It is much easier to analyze and manipulate this intermediate form of the program for the purposes of code optimization than it would be to perform the corresponding operations on either the source program or the machine code.

In Section 5.2.1 we introduce one common way of representing a program in such an intermediate form. The intermediate form used in a compiler, if one is used, is not strictly dependent on the machine for which the compiler is designed. However, such a form is necessary for our discussion of machine-dependent code optimization in Section 5.2.2. There are also many machine-independent techniques for code optimization that use a similar intermediate representation of the program. Some of these techniques are discussed in Section 5.3.

5.2.1 Intermediate Form of the Program

There are many possible ways of representing a program in an intermediate form for code analysis and optimization. Discussions of some of these can be found in Aho et al. (1988). The method we describe in this section represents the executable instructions of the program with a sequence of *quadruples*. Each quadruple is of the form

```
operation, op1, op2, result
```

where *operation* is some function to be performed by the object code, *op1* and *op2* are the operands for this operation, and *result* designates where the resulting value is to be placed.

For example, the source program statement

```
SUM := SUM + VALUE
```

could be represented with the quadruples

```
+ ,   SUM, VALUE, i1
```

```
:=,   i1 ,       , SUM
```

The entry i_1 designates an intermediate result (SUM + VALUE); the second quadruple assigns the value of this intermediate result to SUM. Assignment is treated as a separate operation (:=) to open up additional possibilities for code optimization. Similarly, the statement

```
VARIANCE := SUMSQ DIV 100 - MEAN * MEAN
```

could be represented with the quadruples

```
DIV,  SUMSQ,  #100,     i1
*  ,  MEAN ,  MEAN,     i2
-  ,  i1   ,  i2  ,     i3
:= ,  i3   ,      ,     VARIANCE
```

These quadruples would be created by intermediate code-generation routines similar to those discussed in Section 5.1.4. Many types of analysis and manipulation can be performed on the quadruples for code-optimization purposes. For example, the quadruples can be rearranged to eliminate redundant load and store operations, and the intermediate results i_j can be assigned to registers or to temporary variables to make their use as efficient as possible. We discuss some of these possibilities in later sections. After optimization has been performed, the modified quadruples are translated into machine code.

Note that the quadruples appear in the order in which the corresponding object code instructions are to be executed. This greatly simplifies the task of analyzing the code for purposes of optimization. It also means that the translation into machine instructions will be relatively easy.

Figure 5.18 shows a sequence of quadruples corresponding to the source program in Fig. 5.1. Note that the READ and WRITE statements are represented with a CALL operation, followed by PARAM quadruples that specify the parameters of the READ or WRITE. These PARAM quadruples will, of course, be translated into parameter list entries, like those shown in Fig. 5.17, when the machine code is generated. The JGT operation in quadruple 4 compares the values of its two operands and jumps to quadruple 15 if the first operand is greater than the second. The J operation in quadruple 14 jumps unconditionally to quadruple 4.

You should compare Figs. 5.18 and 5.1 to see the correspondence between source statements and quadruples. You may also want to compare the quadruples in Fig. 5.18 with the object code representation shown in Fig. 5.17.

	Operation	Op1	Op2	Result	
(1)	:=	#0		SUM	{SUM := 0}
(2)	:=	#0		SUMSQ	{SUMSQ := 0}
(3)	:=	#1		I	{FOR I := 1 TO 100}
(4)	JGT	I	#100	(15)	
(5)	CALL	XREAD			{READ(VALUE)}
(6)	PARAM	VALUE			
(7)	+	SUM	VALUE	i_1	{SUM := SUM + VALUE}
(8)	:=	i_1		SUM	
(9)	*	VALUE	VALUE	i_2	{SUMSQ := SUMSQ +
(10)	+	SUMSQ	i_2	i_3	VALUE * VALUE}
(11)	:=	i_3		SUMSQ	
(12)	+	I	#1	i_4	{end of FOR loop}
(13)	:=	i_4		I	
(14)	J			(4)	
(15)	DIV	SUM	#100	i_5	{MEAN := SUM DIV 100}
(16)	:=	i_5		MEAN	
(17)	DIV	SUMSQ	#100	i_6	{VARIANCE :=
(18)	*	MEAN	MEAN	i_7	SUMSQ DIV 100
(19)	-	i_6	i_7	i_8	- MEAN * MEAN}
(20)	:=	i_8		VARIANCE	
(21)	CALL	XWRITE			{WRITE(MEAN,VARIANCE)}
(22)	PARAM	MEAN			
(23)	PARAM	VARIANCE			

Figure 5.18 Intermediate code for the program from Fig. 5.1.

5.2.2 Machine-Dependent Code Optimization

In this section we briefly describe several different possibilities for performing machine-dependent code optimization. Further details concerning these techniques can be found in many textbooks on compilers, such as Aho et al. (1988).

The first problem we discuss is the assignment and use of registers. On many computers there are a number of general-purpose registers that may be used to hold constants, the values of variables, intermediate results, and so on. These same registers can also often be used for addressing (as base or index registers). We concentrate here, however, on the use of registers as instruction operands.

Machine instructions that use registers as operands are usually faster than the corresponding instructions that refer to locations in memory. Therefore, we would prefer to keep in registers all variables and intermediate results that will be used later in the program. Each time a value is fetched from memory, or calculated as an intermediate result, it can be assigned to some register. The value will be available for later use without requiring a memory reference.

This approach also avoids unnecessary movement of values between memory and registers, which takes time but does not advance the computation. We used a very simple version of this technique in Section 5.1.4 when we kept track of the value currently in register A.

Consider, for example, the quadruples shown in Fig. 5.18. The variable VALUE is used once in quadruple 7 and twice in quadruple 9. If enough registers are available, it would be possible to fetch this value only once. The value would be retained in a register for use by the code generated from quadruple 9. Likewise, quadruple 16 stores the value of i_5 into the variable MEAN. If i_5 is assigned to a register, this value could still be available when the value of MEAN is required in quadruple 18. Such register assignments can also be used to eliminate much of the need for temporary variables. Consider, for example, the machine code in Fig. 5.17, in which the use of only one register (register A) was sufficient to handle six of the eight intermediate results (i_j) in Fig. 5.18.

Of course, there are rarely as many registers available as we would like to use. The problem then becomes one of selecting which register value to replace when it is necessary to assign a register for some other purpose. One reasonable approach is to scan the program for the next point at which each register value would be used. The value that will not be needed for the longest time is the one that should be replaced. If the register that is being reassigned contains the value of some variable already stored in memory, the value can simply be discarded. Otherwise, this value must be saved using a temporary variable. This is one of the functions performed by the GETA procedure discussed in Section 5.1.4, using the temporary variables Ti.

In making and using register assignments, a compiler must also consider the control flow of the program. For example, quadruple 1 in Fig. 5.18 assigns the value 0 to SUM. This value might be retained in some register for later use. When SUM is next used as an operand in quadruple 7, it might appear that its value can be taken directly from the register; however, this is not necessarily the case. The J operation in quadruple 14 jumps to quadruple 4. If control passes to quadruple 7 in this way, the value of SUM may not be in the designated register. This would happen, for example, if the register were reassigned to hold the value of i_4 in quadruple 12. Thus the existence of Jump instructions creates difficulty in keeping track of register contents.

One way to deal with this problem is to divide the program into *basic blocks*. A basic block is a sequence of quadruples with one entry point, which is at the beginning of the block, one exit point, which is at the end of the block, and no jumps within the block. In other words, each quadruple that is the target of a jump, or that immediately follows a quadruple that specifies a jump, begins a new basic block. Since procedure calls can have unpredictable effects on register contents, a CALL operation is also usually considered to begin a new basic block. The assignment and use of registers within a basic block can follow the method

previously described. However, when control passes from one basic block to another, all values currently held in registers are saved in temporary variables.

Figure 5.19 shows the division of the quadruples from Fig. 5.18 into basic blocks. Block A contains quadruples 1–3; block B contains quadruple 4, and so on. Figure 5.19 also shows a representation of the control flow of the program. An arrow from block X to block Y indicates that control can pass directly from the last quadruple of X to the first quadruple of Y. (Within a basic block, of course, control must pass sequentially from one quadruple to the next.) This kind of representation is called a *flow graph* for the program. More sophisticated code-optimization techniques can analyze a flow graph and perform register assignments that remain valid from one basic block to another.

Another possibility for code optimization involves rearranging quadruples before machine code is generated. Consider, for example, the quadruples in Fig. 5.20(a). These are essentially the same as quadruples 17–20 in Fig. 5.18; they correspond to source statement 14 of the program in Fig. 5.1. Figure 5.20(a) shows a typical generation of machine code from these quadruples, using only a single register (register A). This is the same as the code shown for source statement 14 in Fig. 5.17.

Note that the value of the intermediate result i_1 is calculated first and stored in temporary variable T1. Then the value of i_2 is calculated. The third quadruple in this series calls for subtracting the value of i_2 from i_1. Since i_2 has just been computed, its value is available in register A; however, this does no good, since the *first* operand for a – operation must be in the register. It is necessary to store the value of i_2 in another temporary variable, T2, and then load the value of i_1 from T1 into register A before performing the subtraction.

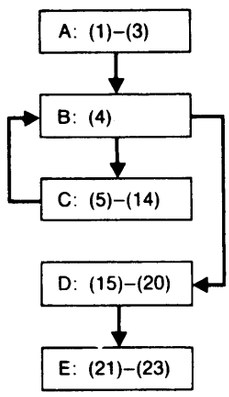


Figure 5.19 Basic blocks and flow graph for the quadruples from Fig. 5.18.

```
DIV   SUMSQ    #100     i1
*     MEAN     MEAN     i2
-     i1       i2       i3
:=    i3                VARIANCE
```

```
LDA     SUMSQ
DIV     #100
STA     T1
LDA     MEAN
MUL     MEAN
STA     T2
LDA     T1
SUB     T2
STA     VARIANCE
```

(a)

```
*     MEAN     MEAN     i2
DIV   SUMSQ    #100     i1
-     i1       i2       i3
:=    i3                VARIANCE
```

```
LDA     MEAN
MUL     MEAN
STA     T1
LDA     SUMSQ
DIV     #100
SUB     T1
STA     VARIANCE
```

(b)

Figure 5.20 Rearrangement of quadruples for code optimization.

With a little analysis, an optimizing compiler could recognize this situation and rearrange the quadruples so the second operand of the subtraction is computed first. This rearrangement is illustrated in Fig. 5.20(b). The first two quadruples in the sequence have been interchanged. The resulting machine code requires two fewer instructions and uses only one temporary

variable instead of two. The same technique can be applied to rearrange quadruples that calculate the operands of a DIV operation or any other operation for which the machine code requires a particular placement of operands.

Other possibilities for machine-dependent code optimization involve taking advantage of specific characteristics and instructions of the target machine. For example, there may be special loop-control instructions or addressing modes that can be used to create more efficient object code. On some computers there are high-level machine instructions that can perform complicated functions such as calling procedures and manipulating data structures in a single operation. Obviously the use of such features, when possible, can greatly improve the efficiency of the object program.

Some machines have a CPU that is made up of several functional units. On such computers, the order in which machine instructions appear can affect the speed of execution. Consecutive instructions that involve different functional units can sometimes be executed at the same time. An optimizing compiler for such a machine could rearrange object code instructions to take advantage of this property. For examples and references, see Aho et al. (1988).

5.3 MACHINE-INDEPENDENT COMPILER FEATURES

In this section we briefly describe some common compiler features that are largely independent of the particular machine being used. As in the preceding section, we do not attempt to give full details of the implementation of these features. Such details may be found in the references cited.

Section 5.3.1 describes methods for handling structured variables such as arrays. Section 5.3.2 continues the discussion of code optimization begun in Section 5.2.2. This time, we are concerned with machine-independent techniques for optimizing the object code.

In the compiler design described in Section 5.1, we dealt only with simple variables that were permanently assigned to storage locations within the object program. Section 5.3.3 describes some alternative ways of performing storage allocation for the compiled program. Section 5.3.4 discusses the problems involved in compiling a block-structured language and indicates some possible solutions for these problems.

5.3.1 Structured Variables

In this section we briefly consider the compilation of programs that use *structured variables* such as arrays, records, strings, and sets. We are primarily concerned

with the allocation of storage for such variables and with the generation of code to reference them. These issues are discussed in a moderate amount of detail for arrays. The same principles can also be applied to the other types of structured variables. Further details concerning these topics can be found in a number of textbooks on compilers, such as Aho et al. (1988).

Consider first the Pascal array declaration

```
A : ARRAY[1..10] OF INTEGER
```

If each INTEGER variable occupies one word of memory, then we must clearly allocate ten words to store this array. More generally, if an array is declared as

ARRAY[*l*..*u*] OF INTEGER

then we must allocate $u - l + 1$ words of storage for the array.

Allocation for a multi-dimensional array is not much more difficult. Consider, for example, the two-dimensional array

```
B : ARRAY[0..3, 1..6] OF INTEGER
```

Here the first subscript can take on four different values (0–3), and the second subscript can take on six different values. We need to allocate a total of 4 * 6 = 24 words to store the array. In general, if the array declaration is

ARRAY [l_1..u_1, l_2..u_2] OF INTEGER

then the number of words to be allocated is given by

$$(u_1 - l_1 + 1) * (u_2 - l_2 + 1)$$

For an array with n dimensions, the number of words required is a product of n such terms.

When we consider the generation of code for array references, it becomes important to know which array element corresponds to each word of allocated storage. For one-dimensional arrays, there is an obvious correspondence. In the array A previously defined, the first word would contain A[1], the second word would contain A[2], and so on. For higher-dimensional arrays, however, the choice of representation is not as clear.

Figure 5.21 shows two possible ways of storing the previously defined array B. In Fig. 5.21(a), all array elements that have the same value of the first subscript are stored in contiguous locations; this is called *row-major* order. In Fig. 5.21(b), all elements that have the same value of the second subscript are

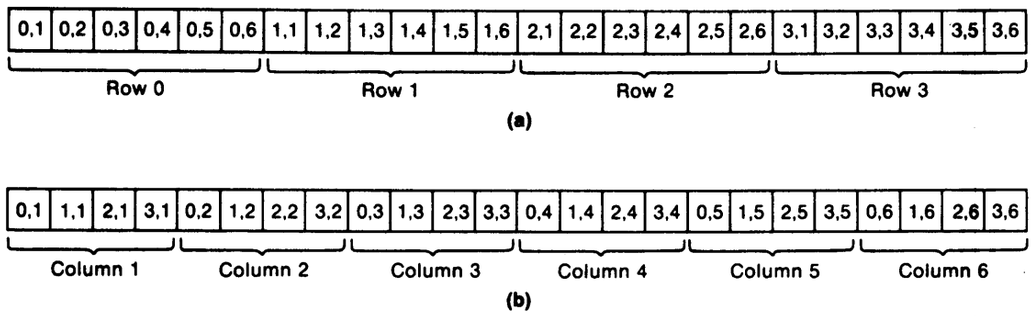


Figure 5.21 Storage of B : ARRAY[0..3, 1..6] in (a) row-major order and (b) column-major order.

stored together; this is called *column-major* order. Another way of looking at this is to scan the words of the array in sequence and observe the subscript values. In row-major order, the rightmost subscript varies most rapidly; in column-major order, the leftmost subscript varies most rapidly. These concepts can be generalized easily to arrays with more than two subscripts.

Compilers for most high-level languages store arrays using row-major order; this is the order we assume in the following discussions. For historical reasons, however, most FORTRAN compilers store arrays in column-major order.

To refer to an array element, we must calculate the address of the referenced element relative to the base address of the array. For a typical computer, the compiler would generate code to place this relative address in an index register. Indexed addressing would then be used to access the desired array element. In the following discussion we assume the base address is the address of the first word allocated to store the array. For a discussion of other possibilities, see Aho et al. (1988).

Consider first the one-dimensional array

```
A : ARRAY[1..10] OF INTEGER
```

and suppose that a statement refers to array element A[6]. There are five array elements preceding A[6]; on a SIC machine, each such element would occupy 3 bytes. Thus the address of A[6] relative to the starting address of the array is given by 5 * 3 = 15.

If an array reference involves only constant subscripts, the relative address calculation can be performed during compilation. If the subscripts involve variables, however, the compiler must generate object code to perform this calculation during execution. Suppose the array declaration is

```
A : ARRAY[l..u] OF INTEGER
```

and each array element occupies w bytes of storage. If the value of the subscript is s, then the relative address of the referenced array element A[s] is given by

$$w * (s - l)$$

The generation of code to perform such a calculation is illustrated in Fig. 5.22(a). The notation $A[i_2]$ in quadruple 3 specifies that the generated machine code should refer to A using indexed addressing, after having placed the value of i_2 in the index register.

For multi-dimensional arrays, the generation of code depends on whether row-major or column-major order is used to store the array. We assume row-major order. Figure 5.21(a) illustrates the storage of the array

```
B : ARRAY [0..3, 1..6] OF INTEGER
```

in row-major order. Consider first the array element B[2,5]. If we start at the beginning of the array, we must skip over two complete rows (row 0 and row 1) before arriving at the beginning of row 2 (i.e., element B[2,1]). Each such row contains six elements, so this involves 2 * 6 = 12 array elements. We must also skip over the first four elements in row 2 to arrive at B[2,5]. This makes a total of 16 array elements between the beginning of the array and element B[2,5]. If each element occupies 3 bytes, then B[2,5] is located at relative address 48 within the array.

More generally, suppose the array declaration is

```
B : ARRAY [l1..u1, l2..u2] OF INTEGER
```

and we wish to refer to an array element specified by subscripts having values s_1 and s_2. The relative address of $B[s_1,s_2]$ is given by

$$w * [(s_1 - l_1) * (u_2 - l_2 + 1) + (s_2 - l_2)]$$

Figure 5.22(b) illustrates the generation of code to perform such an array reference. You should examine this set of quadruples carefully to be sure you understand the calculations involved.

The methods and formulas discussed above can easily be generalized to higher-dimensional arrays. For details, see Aho et al. (1988).

The symbol-table entry for an array usually specifies the type of the elements in the array, the number of dimensions declared, and the lower and upper limit for each subscript. This information is sufficient for the compiler to

```
A : ARRAY [1..10] OF INTEGER
        .
        .
        .
A[I]    := 5
```

```
(1)     -       I       #1      i1
(2)     *       i1      #3      i2
(3)     :=      #5              A[i2]
```

(a)

```
B : ARRAY [0..3,1..6] OF INTEGER
        .
        .
        .
B[I,J]  := 5
```

```
(1)     *       I       #6      i1
(2)     -       J       #1      i2
(3)     +       i1      i2      i3
(4)     *       i3      #3      i4
(5)     :=      #5              B[i4]
```

(b)

Figure 5.22 Code generation for array references.

generate the code required for array references. In some languages, however, the required information may not be known at compilation time. For example, FORTRAN 90 provides *dynamic arrays*. Using this feature, a two-dimensional array could be declared as

```
INTEGER, ALLOCABLE, ARRAY (:,:) :: MATRIX
```

This specifies that MATRIX is an array of integers that can be allocated dynamically. The allocation can be accomplished by a statement like

```
ALLOCATE (MATRIX(ROWS,COLUMNS))
```

where the variables ROWS and COLUMNS have previously been assigned values.

Since the values of ROWS and COLUMNS are not known at compilation time, the compiler cannot directly generate code like that in Fig. 5.22. Instead, the compiler creates a descriptor (often called a *dope vector)* for the array. This descriptor includes space for storing the lower and upper bounds for each array subscript. When storage is allocated for the array, the values of these bounds are computed and stored in the descriptor. The generated code for an array reference uses the values from the descriptor to calculate relative addresses as required. The descriptor may also include the number of dimensions for the array, the type of the array elements, and a pointer to the beginning of the array. This information can be useful if the allocated array is passed as a parameter to another procedure.

The issues discussed for arrays also arise in the compilation of other structured variables such as records, strings, and sets. The compiler must provide for the allocation of storage for the variable; it must store information concerning the structure of the variable, and use this information to generate code to access components of the structure; and it must construct a descriptor for situations in which the required information is not known at compilation time. For further discussion of these issues as they relate to specific types of structured variables, see Aho et al. (1988) and Fischer and LeBlanc (1988).

5.3.2 Machine-Independent Code Optimization

In this section we discuss some of the most important types of machine-independent code optimization. As in the previous sections, we do not attempt to give the full details of any of these techniques. Instead, we give an intuitive verbal description and illustrate the main concepts with examples. Algorithms and further details concerning these methods can be found in Aho et al. (1988). We assume the source program has already been translated into a sequence of quadruples like those introduced in Section 5.2.1.

One important source of code optimization is the elimination of *common subexpressions.* These are subexpressions that appear at more than one point in the program and that compute the same value. Consider, for example, the statements in Fig. 5.23(a). The term 2*J is a common subexpression. An optimizing compiler should generate code so that this multiplication is performed only once and the result is used in both places.

```
X,Y : ARRAY[1..10,1..10] OF INTEGER
      .
      .
      .
FOR I := 1 TO 10 DO
   X[I,2*J-1] := Y[I,2*J]
```

(a)

(1)	:=	#1		I	{loop initialization}
(2)	JGT	I	#10	(20)	
(3)	-	I	#1	i_1	{subscript calculation for X}
(4)	*	i_1	#10	i_2	
(5)	*	#2	J	i_3	
(6)	-	i_3	#1	i_4	
(7)	-	i_4	#1	i_5	
(8)	+	i_2	i_5	i_6	
(9)	*	i_6	#3	i_7	
(10)	-	I	#1	i_8	{subscript calculation for Y}
(11)	*	i_8	#10	i_9	
(12)	*	#2	J	i_{10}	
(13)	-	i_{10}	#1	i_{11}	
(14)	+	i_9	i_{11}	i_{12}	
(15)	*	i_{12}	#3	i_{13}	
(16)	:=	$Y[i_{13}]$		$X[i_7]$	{assignment operation}
(17)	+	#1	I	i_{14}	{end of loop}
(18)	:=	i_{14}		I	
(19)	J			(2)	
(20)					{next statement}

(b)

(1)	:=	#1		I	{loop initialization}
(2)	JGT	I	#10	(16)	
(3)	-	I	#1	i_1	{subscript calculation for X}
(4)	*	i_1	#10	i_2	
(5)	*	#2	J	i_3	
(6)	-	i_3	#1	i_4	
(7)	-	i_4	#1	i_5	
(8)	+	i_2	i_5	i_6	
(9)	*	i_6	#3	i_7	
(10)	+	i_2	i_4	i_{12}	{subscript calculation for Y}
(11)	*	i_{12}	#3	i_{13}	
(12)	:=	$Y[i_{13}]$		$X[i_7]$	{assignment operation}
(13)	+	#1	I	i_{14}	{end of loop}
(14)	:=	i_{14}		I	
(15)	J			(2)	
(16)					{next statement}

(c)

Figure 5.23 Code optimization by elimination of common subexpressions and removal of loop invariants.

```
(1)   *      #2          J       i3        {computation of invariants}
(2)   -      i3          #1      i4
(3)   -      i4          #1      i5
(4)   :=     #1                  I         {loop initialization}
(5)   JGT    I           #10     (16)
(6)   -      I           #1      i1        {subscript calculation for X}
(7)   *      i1          #10     i2
(8)   +      i2          i5      i6
(9)   *      i6          #3      i7
(10)  +      i2          i4      i12       {subscript calculation for Y}
(11)  *      i12         #3      i13
(12)  :=     Y[i13]              X[i7]     {assignment operation}
(13)  +      #1          I       i14       {end of loop}
(14)  :=     i14                 I
(15)  J                          (5)
(16)                                       {next statement}
```

(d)

Figure 5.23 (*cont'd*)

Common subexpressions are usually detected through the analysis of an intermediate form of the program. Such an intermediate form is shown in Fig. 5.23(b). If we examine this sequence of quadruples, we see that quadruples 5 and 12 are the same except for the name of the intermediate result produced. Note that the operand J is not changed in value between quadruples 5 and 12. It is not possible to reach quadruple 12 without passing through quadruple 5 first because the quadruples are part of the same basic block. Therefore, quadruples 5 and 12 compute the same value. This means we can delete quadruple 12 and replace any reference to its result (i_{10}) with a reference to i_3, the result of quadruple 5. This modification eliminates the duplicate calculation of 2*J, which we identified previously as a common subexpression in the source statement.

After the substitution of i_3 for i_{10} is performed, quadruples 6 and 13 are the same, except for the name of the result. Thus we can remove quadruple 13 and substitute i_4 for i_{11} wherever it is used. Similarly, quadruples 10 and 11 can be removed because they are equivalent to quadruples 3 and 4.

The result of applying this technique is shown in Fig. 5.23(c). The quadruples have been renumbered in this figure. However, the intermediate result names i_j have been left unchanged, except for the substitutions just described, to make the comparison with Fig. 5.23(b) easier. Note that the total number of quadruples has been reduced from 19 to 15. Each of the quadruple operations used here will probably take approximately the same length of time to execute on a typical machine, so there should be a corresponding reduction in the overall execution time of the program.

Another common source of code optimization is the removal of *loop invariants*. These are subexpressions within a loop whose values do not change from one iteration of the loop to the next. Thus their values can be computed once, before the loop is entered, rather than being recalculated for each iteration. Because most programs spend most of their running time in the execution of loops, the time savings from this sort of optimization can be highly significant. We assume the existence of algorithms that can detect loops by analyzing the control flow of the program. One example of such an algorithm is the method for constructing a program flow graph that is described in Section 5.2.2.

An example of a loop-invariant computation is the term 2*J in Fig. 5.23(a) [see quadruple 5 of Fig. 5.23(c)]. The result of this computation depends only on the operand J, which does not change in value during the execution of the loop. Thus we can move quadruple 5 in Fig. 5.23(c) to a point immediately before the loop is entered. A similar argument can be applied to quadruples 6 and 7.

Figure 5.23(d) shows the sequence of quadruples that results from these modifications. The total number of quadruples remains the same as in Fig. 5.23(c); however, the number of quadruples within the body of the loop has been reduced from 14 to 11. Each execution of the FOR statement in Fig. 5.23(a) causes 10 iterations of the loop, which means the total number of quadruple operations required for one execution of the FOR is reduced from 141 to 114.

Our modifications have reduced the total number of quadruple operations for one execution of the FOR from 181 [Fig. 5.23(b)] to 114 [Fig. 5.23(d)], which saves a substantial amount of time. There are methods for handling common subexpressions and loop invariants that are considerably more sophisticated than the techniques we used here. As might be expected, such methods can produce more highly optimized code. For examples and discussions of these, see Aho et al. (1988).

Some optimization, of course, can be obtained by rewriting the source program. For example, the statements in Fig. 5.23(a) could have been written as

```
T1  := 2 * J;
T2  := T1 - 1;
FOR I := 1 TO 10 DO
  X[I,T2] := Y[I,T1]
```

However, this would achieve only a part of the benefits realized by the optimization process just described. The rest of the optimizations are related to the process of calculating a relative address from subscript values; these details are inaccessible to the source programmer. For example, the optimizations involving quadruples 3, 4, 10, and 11 in Fig. 5.23(b) could not be achieved with any rewriting of the source statement. It could also be argued that the original

statements in Fig. 5.23(a) are preferable because they are clearer than the modified version involving T1 and T2. An optimizing compiler should allow the programmer to write source code that is clear and easy to read, and it should compile such a program into machine code that is efficient to execute.

Another source of code optimization is the substitution of a more efficient operation for a less efficient one. Consider, for example, the FORTRAN program segment in Fig. 5.24(a). This DO loop creates a table that contains the first 20 powers of 2. In each iteration of the loop, the constant 2 is raised to the power I. Figure 5.24(b) shows a representation of these statements as a series of quadruples. Exponentiation is represented with the operation EXP. At the machine-code level, EXP might involve either a loop that performs a series of multiplications, or a call to a subroutine that uses logarithms to arrive at the result.

```
      DO 10 I = 1,20
10    TABLE(I) = 2**I
```

(a)

(1)	:=	#1		I	{loop initialization}
(2)	EXP	#2	I	i_1	{calculation of 2**I}
(3)	-	I	#1	i_2	{subscript calculation}
(4)	*	i_2	#3	i_3	
(5)	:=	i_1		TABLE[i_3]	{assignment operation}
(6)	+	I	#1	i_4	{end of loop}
(7)	:=	i_4		I	
(8)	JLE	I	#20	(2)	

(b)

(1)	:=	#1		i_1	{initialize temporaries}
(2)	:=	#(-3)		i_3	
(3)	:=	#1		I	{loop initialization}
(4)	*	i_1	#2	i_1	{calculation of 2**I}
(5)	+	i_3	#3	i_3	{subscript calculation}
(6)	:=	i_1		TABLE[i_3]	{assignment operation}
(7)	+	I	#1	i_4	{end of loop}
(8)	:=	i_4		I	
(9)	JLE	I	#20	(4)	

(c)

Figure 5.24 Code optimization by reduction in strength of operations.

On closer examination, we can see that there is a more efficient way to perform the computation. For each iteration of the loop, the value of I increases by 1. Therefore, the value of 2**I for the current iteration can be found by multiplying the value for the previous iteration by 2. Clearly this method of computing 2**I is much more efficient than performing a series of multiplications or using a logarithmic technique. Such a transformation is called *reduction in strength* of an operation.

A similar transformation can be applied in the calculation of the relative address for the array element TABLE(I). Assuming that each array element occupies one word, a SIC object program would calcu! te this displacement as 3 * (I – 1). This calculation appears in quadruples 3 and 4 of Fig. 5.24(b). Thus the object program would need to perform one multiplication for each array reference.

The same sort of reduction in strength can be applied to this situation. Each iteration of the loop refers to the next array element in sequence, so the calculation of the required displacement can be accomplished by adding 3 to the previous displacement. If addition is faster than multiplication on the target machine, this reduction in strength will result in more efficient object code.

Figure 5.24(c) shows a modification of the quadruples from Fig. 5.24(b) that implements these two reductions in strength. An algorithm for performing this sort of transformation can be found in Aho et al. (1988). As in our previous examples, a part of this optimization could have been performed at the source-code level. However, the strength reduction in the subscript calculation process could not be accomplished by the programmer, who has no access to the details of code generation for array references.

There are a number of other possibilities for machine-independent code optimization. For example, computations whose operand values are known at compilation time can be performed by the compiler. This optimization is known as *folding*. Other optimizations include converting a loop into straight-line code (*loop unrolling*) and merging of the bodies of loops (*loop jamming*). For details on these and other optimization techniques, see Aho et al. (1988).

5.3.3 Storage Allocation

In the compilation scheme presented in Section 5.1, all programmer-defined variables were assigned storage locations within the object program as their declarations were processed. Temporary variables, including the one used to save the return address, were also assigned fixed addresses within the program. This simple type of storage assignment is usually called *static* allocation. It is often used for languages that do not allow the recursive use of procedures or subroutines, and do not provide for the dynamic allocation of storage during execution.

If procedures may be called recursively, static allocation cannot be used. Consider, for example, Fig. 5.25. In Fig. 5.25(a), the program MAIN has been called by the operating system or the loader (invocation 1). The first action taken by MAIN is to store its return address from register L at a fixed location RETADR within MAIN. In Fig. 5.25(b), MAIN has called the procedure SUB (invocation 2). The return address for this call has been stored at a fixed location within SUB. If SUB now calls itself recursively, as in Fig. 5.25(c), a problem occurs. SUB stores the return address for invocation 3 into RETADR from register L. This destroys the return address for invocation 2. As a result, there is no possibility of ever making a correct return to MAIN.

A similar difficulty occurs with respect to any variables used by SUB. When the recursive call is made, variables within SUB may be set to new values; this destroys the previous contents of these variables. However, these previous values may be needed by invocation 2 of SUB after the return from the recursive call. This is the same problem mentioned in Section 4.3.1 in our discussion of recursive macro expansion.

Obviously it is necessary to preserve the previous values of any variables used by SUB, including parameters, temporaries, return addresses, register

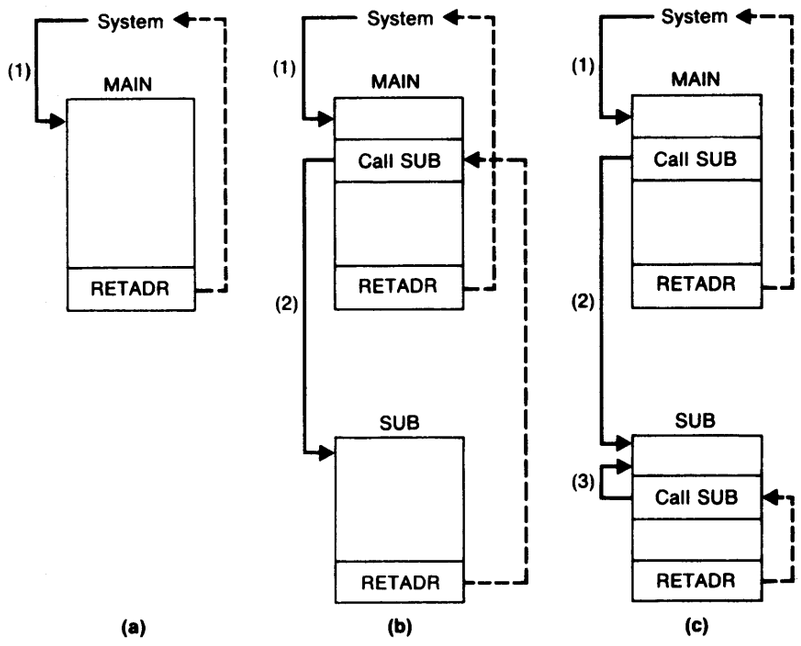


Figure 5.25 Recursive invocation of a procedure using static storage allocation.

save areas, etc., when the recursive call is made. This is usually accomplished with a dynamic storage allocation technique. Each procedure call creates an *activation record* that contains storage for all the variables used by the procedure. If the procedure is called recursively, another activation record is created. Each activation record is associated with a particular *invocation* of the procedure, not with the procedure itself. An activation record is not deleted until a return has been made from the corresponding invocation. The starting address for the current activation record is usually contained in a base register, which is used by the procedure to address its variables. In this way, the values of variables used by different invocations of a procedure are kept separate from one another.

Activation records are typically allocated on a stack, with the current record at the top of the stack. This process is illustrated in Fig. 5.26. In Fig. 5.26(a), which corresponds to Fig. 5.25(a), the procedure MAIN has been called; its activation record appears on the stack. The base register B has been set to indicate the starting address of this current activation record. The first word in

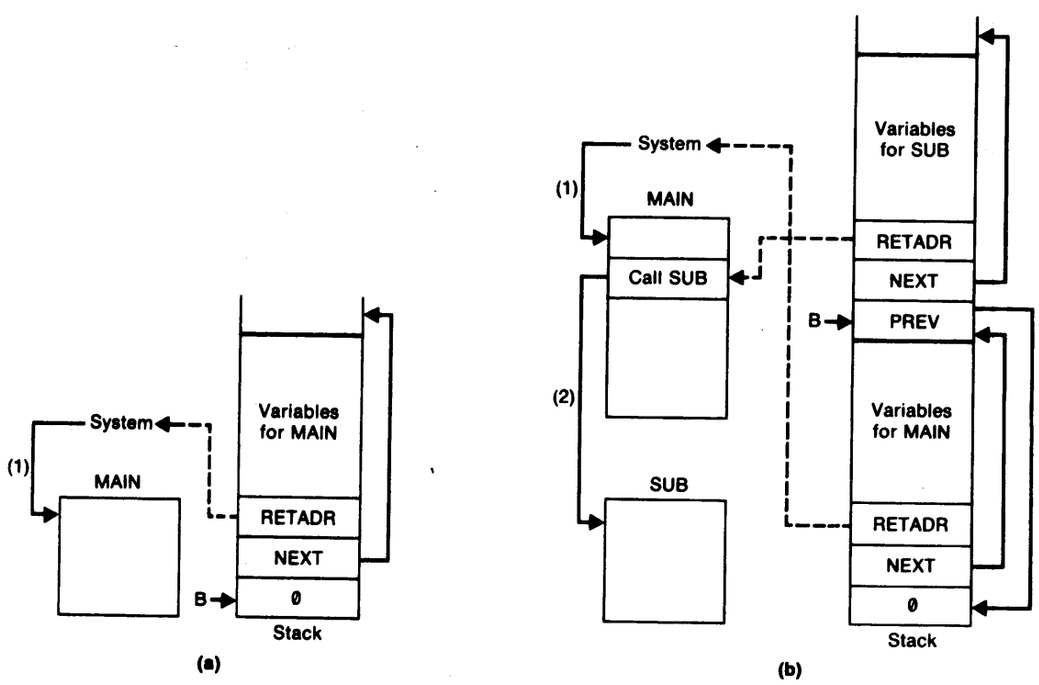


Figure 5.26 Recursive invocation of a procedure using automatic storage allocation.

an activation record would normally contain a pointer PREV to the previous record on the stack. Since this record is the first, the pointer value is null. The second word of the activation record contains a pointer NEXT to the first unused word of the stack, which will be the starting address for the next activation record created. The third word contains the return address for this invocation of the procedure, and the remaining words contain the values of variables used by the procedure.

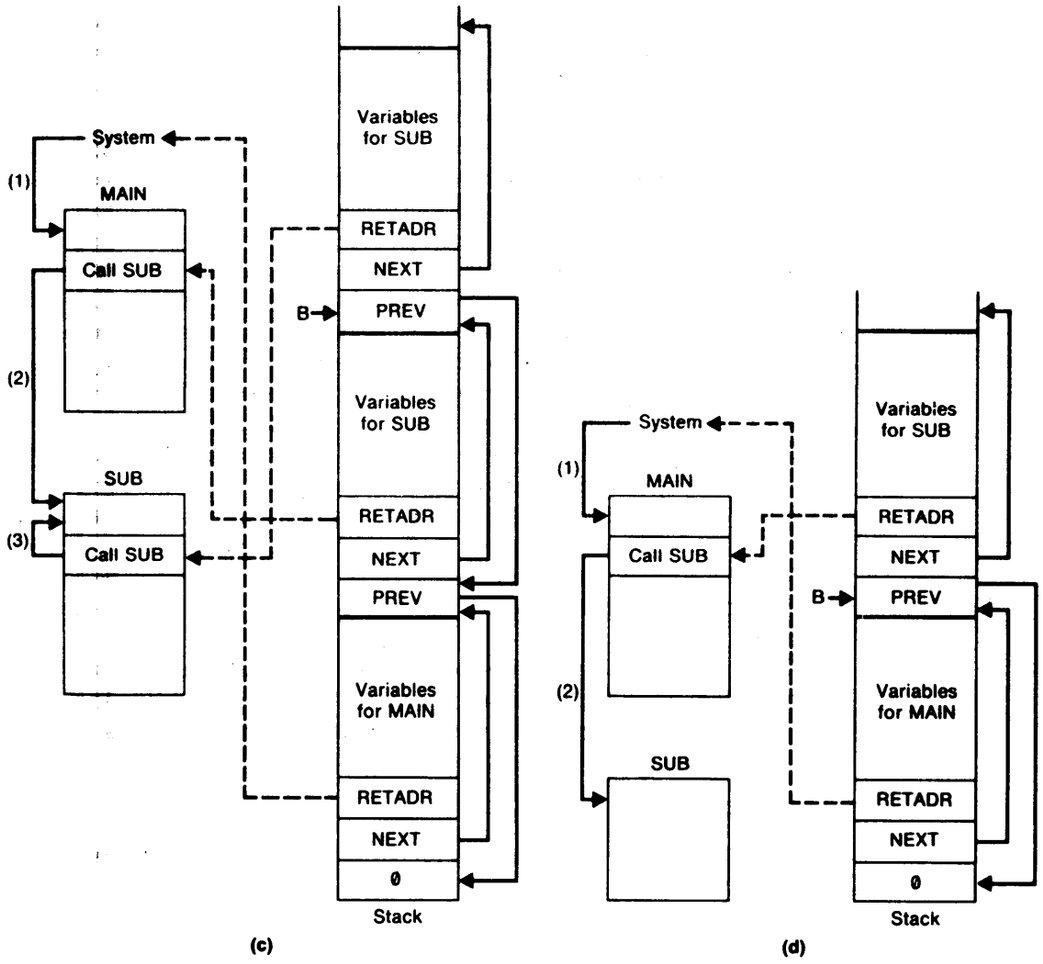


Figure 5.26 (*cont'd*)

In Fig. 5.26(b), MAIN has called the procedure SUB. A new activation record has been created on the top of the stack, with register B set to indicate this new current record. The pointers PREV and NEXT in the two records have been set as shown. In Fig. 5.26(c), SUB has called itself recursively; another activation record has been created for this current invocation of SUB. Note that the return addresses and variable values for the two invocations of SUB are kept separate by this process.

When a procedure returns to its caller, the current activation record (which corresponds to the most recent invocation) is deleted. The pointer PREV in the deleted record is used to reestablish the previous activation record as the current one, and execution continues. Figure 5.26(d) shows the stack as it would appear after SUB returns from the recursive call. Register B has been reset to point to the activation record for the previous invocation of SUB. The return address and all the variable values in this activation record are exactly the same as they were before the recursive call.

This technique is often referred to as *automatic* allocation of storage to distinguish it from other types of dynamic allocation that are under the control of the programmer. When automatic allocation is used, the compiler must generate code for references to variables using some sort of relative addressing. In our example the compiler assigns to each variable an address that is relative to the beginning of the activation record, instead of an actual location within the object program. The address of the current activation record is, by convention, contained in register B, so a reference to a variable is translated as an instruction that uses base relative addressing. The displacement in this instruction is the relative address of the variable within the activation record.

The compiler must also generate additional code to manage the activation records themselves. At the beginning of each procedure there must be code to create a new activation record, linking it to the previous one and setting the appropriate pointers as illustrated in Fig. 5.26. This code is often called a *prologue* for the procedure. At the end of the procedure, there must be code to delete the current activation record, resetting pointers as needed. This code is often called an *epilogue.*

When automatic allocation is used, storage is assigned to all variables used by a procedure when the procedure is called. Other types of dynamic storage allocation allow the programmer to specify when storage is to be assigned. In FORTRAN 90, for example, the statement

```
ALLOCATE (MATRIX(ROWS,COLUMNS))
```

allocates storage for a dynamic array MATRIX with the specified dimensions. The statement

```
DEALLOCATE (MATRIX)
```

releases the storage assigned to MATRIX by a previous ALLOCATE.

Another type of dynamic storage allocation is found in Pascal. The statement

```
NEW(P)
```

allocates storage for a variable and sets the pointer P to indicate the variable just created. The type of the variable created is specified by the way P is declared in the program. The program refers to the created variable by using the pointer P. The statement

```
DISPOSE(P)
```

releases the storage that was previously assigned to the variable pointed to by P. A similar feature is available in C. The function

```
MALLOC(SIZE)
```

allocates a block of storage of the specified size, and returns a pointer to it. The function

```
FREE(P)
```

frees the storage indicated by the pointer P, which was returned by a previous MALLOC.

A variable that is dynamically allocated in this way does not occupy a fixed location in an activation record, so it cannot be referenced directly using base relative addressing. Such a variable is usually accessed using indirect addressing through a pointer variable P. Since P does occupy a fixed location in the activation record, it can be addressed in the usual way.

In the preceding discussions we have not described the mechanism by which storage is allocated for a variable. One approach is to let the operating system handle all storage management. A NEW or MALLOC statement would be translated into a request to the operating system for an area of storage of the required size. Another method is to handle the required allocation through a run-time support procedure associated with the compiler. With this method, a large block of free storage called a *heap* is obtained from the operating system at the beginning of the program. Allocations of storage from the heap are managed by the run-time procedure. In some systems, it is not even necessary for the programmer to free storage explicitly. Instead, a run-time *garbage collection* procedure scans the pointers in the program and reclaims areas from the heap that are no longer being used. Discussion and evaluations of such memory management techniques can be found in Sebesta (1996) and Lewis and Denenberg (1991).

Dynamic storage allocation, as discussed in this section, provides another example of *delayed binding*. The association of an address with a variable is

made when the procedure is executed, not when it is compiled or loaded. This delayed binding allows more flexibility in the use of variables and procedures. However, it also requires more overhead because of the creation of activation records and the use of indirect addressing. (Similar observations were made at the end of Section 3.4.2 with respect to dynamic linking.)

5.3.4 Block-Structured Languages

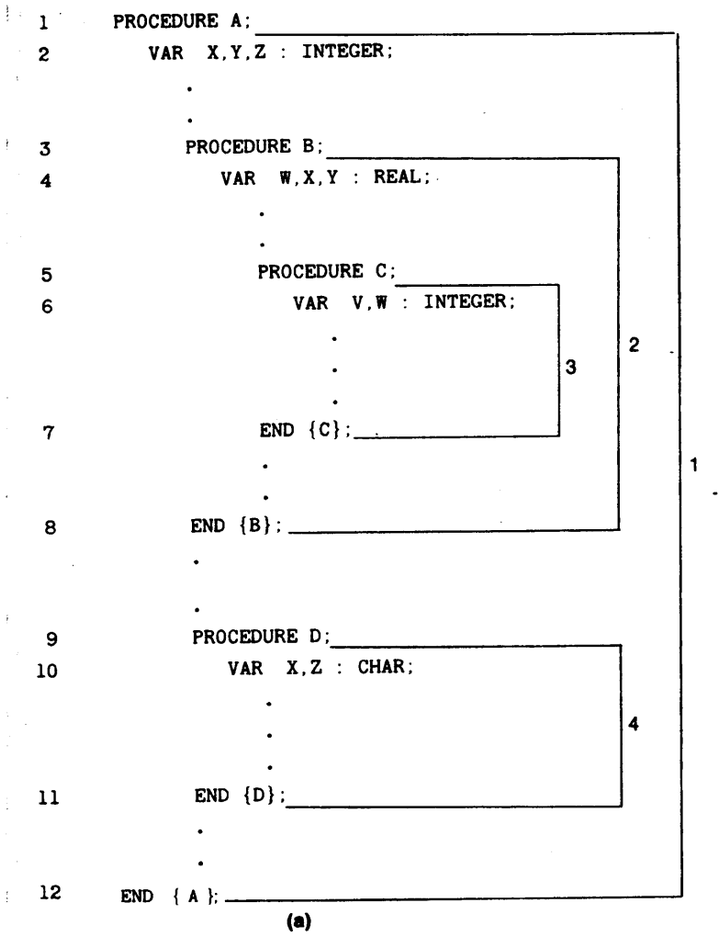
In some languages a program can be divided into units called *blocks*. A block is a portion of a program that has the ability to declare its own identifiers. This definition of a block is also met by units such as procedures and functions in Pascal. In this section we consider some of the issues involved in compiling and executing programs written in such block-structured languages.

Figure 5.27(a) shows the outline of a block-structured program in a Pascal-like language. Each procedure corresponds to a block. In the following discussion, therefore, we use the terms *procedure* and *block* interchangeably. Note that blocks may be nested within other blocks. For example, procedures B and D are nested within procedure A, and procedure C is nested within procedure B. Each block may contain a declaration of variables, as shown. A block may also refer to variables that are defined in any block that contains it, provided the same names are not redefined in the inner block.

Consider, for example, the INTEGER variables X, Y, and Z that are declared in procedure A on line 2. Procedure B contains declarations of X and Y as REAL variables on line 4. Within procedure B, a use of the name X refers to the REAL variable declared within B. However, a use of the name Z refers to the INTEGER variable declared by A because the name Z is not redefined within B. Similarly, within procedure C the name W refers to the variable declared by C; the names X and Y refer to the variables declared by B; and the name Z refers to the variable declared by A. Variables cannot be used outside the block in which they are declared. For example, the name W cannot be referred to outside of procedure B, and V cannot be referred to outside of procedure C.

In compiling a program written in a block-structured language, it is convenient to number the blocks as shown in Fig. 5.27(a). As the beginning of each new block is recognized, it is assigned the next block number in sequence. The compiler can then construct a table that describes the block structure, as illustrated in Fig. 5.27(b). The block-level entry gives the nesting depth for each block. The outermost block has a level number of 1, and each other block has a level number that is one greater than that of the surrounding block.

Since a name can be declared more than once in a program (by different blocks), each symbol-table entry for an identifier must contain the number of the declaring block. A declaration of an identifier is legal if there has been no



(a)

Block name	Block number	Block level	Surrounding block
A	1	1	—
B	2	2	1
C	3	3	2
D	4	2	1

(b)

Figure 5.27 Nesting of blocks in a source program.

previous declaration of that identifier by the current block, so there can be several symbol-table entries for the same name. The entries that represent declarations of the same name by different blocks can be linked together in the symbol table with a chain of pointers.

When a reference to an identifier appears in the source program, the compiler must first check the symbol table for a definition of that identifier by the current block. If no such definition is found, the compiler looks for a definition by the block that surrounds the current one, then by the block that surrounds that, and so on. If the outermost block is reached without finding a definition of the identifier, then the reference is an error.

The search process just described can easily be implemented within a symbol table that uses hashed addressing. The hashing function is used to locate one definition of the identifier. The chain of definitions for that identifier is then searched for the appropriate entry. There are other symbol-table organizations that store the definitions of identifiers according to the nesting of the blocks that define them. This kind of structure can make the search for the proper definition more efficient. See, for example, Aho et al. (1988).

Most block-structured languages make use of automatic storage allocation, as described in Section 5.3.3. That is, the variables that are defined by a block are stored in an activation record that is created each time the block is entered. If a statement refers to a variable that is declared within the current block, this variable is present in the current activation record, so it can be accessed in the usual way. However, it is also possible for a statement to refer to a variable that is declared in some surrounding block. In that case, the most recent activation record for that block must be located to access the variable.

One common method for providing access to variables in surrounding blocks uses a data structure called a *display*. The display contains pointers to the most recent activation records for the current block and for all blocks that surround the current one in the source program. When a block refers to a variable that is declared in some surrounding block, the generated object code uses the display to find the activation record that contains this variable.

The use of a display is illustrated in Fig. 5.28. We assume that procedure A has been invoked by the system, A has then called procedure B, and B has called procedure C. The resulting situation is shown in Fig. 5.28(a). The stack contains activation records for the invocations of A, B, and C. The display contains pointers to the activation records for C and for the surrounding blocks (A and B).

Now let us assume procedure C calls itself recursively. Another activation record for C is created on the stack as a result of this call. Any reference to a variable declared by C should use this most recent activation record; the display pointer for C is changed accordingly. Variables that correspond to the previous invocation of C are not accessible for the moment, so there is no display pointer to this activation record. This situation is illustrated in Fig. 5.28(b).

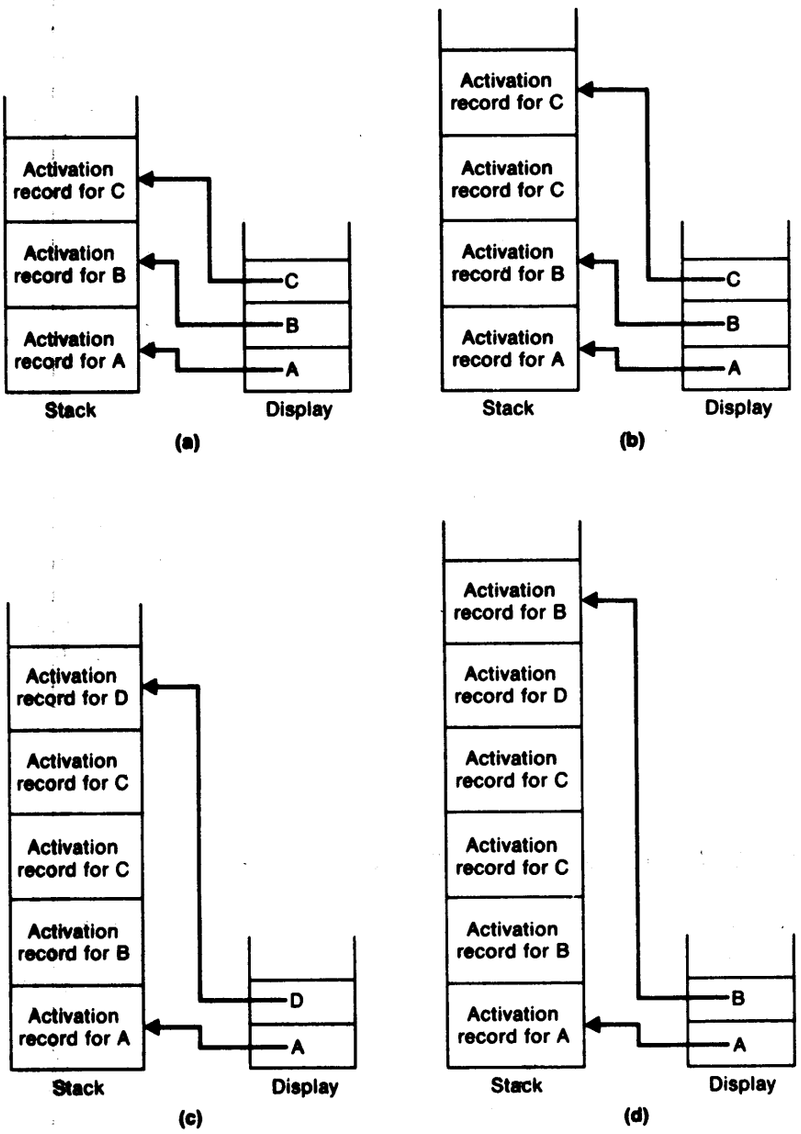


Figure 5.28 Use of display for procedures in Fig. 5.27.

Suppose now that procedure C calls D. (This is allowed because the identifier D is defined in procedure A, which contains C. For simplicity, we have assumed that no special "forward call" declarations are required.) The resulting stack and display are shown in Fig. 5.28(c). An activation record for D has been created in the usual way and added to the stack. Note, however, that the display now contains only two pointers: one each to the activation records for D and A. This is because procedure D cannot refer to variables in B or C, except through parameters that are passed to it, even though it was called from C. According to the rules for the scope of names in a block-structured language, procedure D can refer only to variables that are declared by D or by some block that contains D in the source program (in this case, procedure A).

A similar situation, illustrated in Fig. 5.28(d), occurs if procedure D now calls B. Procedure B is allowed to refer only to variables declared by either B or A, which is reflected in the contents of the display. After procedure B returns to D, the contents of the stack and display will again appear as they were in Fig. 5.28(c).

It is important to be aware of the difference between the run-time allocation of variables, as represented by the stack of activation records, and the rules for referring to variables in the block-structured program, as represented by the display. You should carefully examine Figs. 5.27 and 5.28 to be sure you understand why the stack and display appear as they do in each situation.

The compiler for a block-structured language must include code at the beginning of a block to initialize the display for that block. At the end of the block, it must include code to restore the previous display contents. For the details of how this is accomplished, see Aho et al. (1988).

5.4 COMPILER DESIGN OPTIONS

In this section we consider some of the possible alternatives for the design and construction of a compiler. The discussions in this section are necessarily very brief. Our purpose is to introduce terms and concepts rather than to give a comprehensive discussion of any of these topics.

The compilation scheme presented in Section 5.1 was a simple one-pass design. Sections 5.2 and 5.3 described many features that usually require more than one pass to implement. In Section 5.4.1 we briefly discuss the general question of dividing a compiler into passes, and consider the advantages of one-pass and multi-pass designs.

Section 5.4.2 discusses interpreters, which execute an intermediate form of the program instead of translating it into machine code. Section 5.4.3 introduces the related topic of P-code systems, which compile high-level language programs into object code for a hypothetical machine.

Finally, Section 5.4.4 describes compiler-writing systems, which use software tools to automate much of the process of compiler construction.

5.4.1 Division into Passes

In Section 5.1 we presented a simple one-pass compilation scheme for a subset of the Pascal language. In this design, the compiler was driven by the parsing process. The lexical scanner was called when the parser needed another input token, and a code-generation routine was invoked as each language construct was recognized by the parser. The object code produced was not highly efficient. Most of the code-optimization techniques discussed in Sections 5.2 and 5.3 could not be applied in such a one-pass compiler. However, the compilation process itself, which required only one pass over the program and no intermediate code-generation step, was quite efficient.

Not all languages can be translated by such a one-pass compiler. In Pascal, declarations of variables must appear in the program before the statements that use these variables. In FORTRAN, declarations may appear at the beginning of the program; any variable that is not declared is assigned characteristics by default. However, in some languages the declaration of an identifier may appear after it has been used in the program. One-pass compilers must have the ability to fix up forward references in jump instructions, using techniques like those discussed for one-pass assemblers. Forward references to data items, however, present a much more serious problem.

Consider, for example, the assignment statement

```
X := Y * Z
```

If all of the variables X, Y, and Z are of type INTEGER, the object code for this statement might consist of a simple integer multiplication followed by storage of the result. If the variables are a mixture of REAL and INTEGER types, one or more conversion operations will need to be included in the object code, and floating-point arithmetic instructions may be used. Obviously the compiler cannot decide what machine instructions to generate for this statement unless information about the operands is available. The statement may even be illegal for certain combinations of operand types. Thus a language that allows forward references to data items cannot be compiled in one pass.

Some programming languages, because of other characteristics, require more than two passes to compile. For example Hunter (1981) showed that ALGOL 68 required at least three passes.

There are a number of factors that should be considered in deciding between one-pass and multi-pass compiler designs (assuming that the language in question can be compiled in one pass). If speed of compilation is important,

a one-pass design might be preferred. For example, computers running student jobs tend to spend a large amount of time performing compilations. The resulting object code is usually executed only once or twice for each compilation; these test runs are normally very short. In such an environment, improvements in the speed of compilation can lead to significant benefits in system performance and job turnaround time.

If programs are executed many times for each compilation, or if they process large amounts of data, then speed of execution becomes more important than speed of compilation. In such a case, we might prefer a multi-pass compiler design that could incorporate sophisticated code-optimization techniques. Multi-pass compilers are also used when the amount of memory, or other system resources, is severely limited. The requirements of each pass can be kept smaller if the work of compilation is divided into several passes.

Other factors may also influence the design of the compiler. If a compiler is divided into several passes, each pass becomes simpler and, therefore, easier to understand, write, and test. Different passes can be assigned to different programmers and can be written and tested in parallel, which shortens the overall time required for compiler construction.

For further discussion of the problem of dividing a compiler into passes, see Hunter (1981) and Aho et al. (1988).

5.4.2 Interpreters

An *interpreter* processes a source program written in a high-level language, just as a compiler does. The main difference is that interpreters execute a version of the source program directly, instead of translating it into machine code.

An interpreter usually performs lexical and syntactic analysis functions like those we have described for a compiler, and then translates the source program into an internal form. Many different internal forms can be used. One possibility is a sequence of quadruples like those discussed in Section 5.2. It is even possible to use the original source program itself as the internal form; however, it is generally much more efficient to perform some preprocessing of the program before execution.

After translating the source program into an internal form, the interpreter executes the operations specified by the program. During this phase, an interpreter can be viewed as a set of subroutines. The execution of these subroutines is driven by the internal form of the program.

The process of translating a source program into some internal form is simpler and faster than compiling it into machine code. However, execution of the translated program by an interpreter is much slower than execution of

the machine code produced by a compiler. Thus an interpreter would not normally be used if speed of execution is important. If speed of translation is of primary concern, and execution of the translated program will be short, then an interpreter may be a good choice.

The real advantage of an interpreter over a compiler, however, is in the debugging facilities that can easily be provided. The symbol table, source line numbers, and other information from the source program are usually retained by the interpreter. During execution, these can be used to produce symbolic dumps of data values, traces of program execution related to the source statements, etc. Thus interpreters are especially attractive in an educational environment where the emphasis is on learning and program testing. Discussions of the implementation of debugging tools in an interpreter can be found in Watt (1993).

Most programming languages can be either compiled or interpreted successfully. However, some languages are particularly well suited to the use of an interpreter. As we have seen, compilers usually generate calls to library routines to perform functions such as I/O and complex conversion operations. For some languages, such as SNOBOL and APL, a large part of the compiled program would consist of calls to such routines. In such cases, an interpreter might be preferred because of its speed of translation. Most of the execution time for the translated program would be consumed by the standard library routines. These routines would be the same, regardless of whether a compiler or an interpreter were used.

Certain languages also have features that lend themselves naturally to interpretation. For example, in some languages the type of a variable can change during the execution of a program. Other languages use *dynamic scoping* instead of the more usual *static scoping* we discussed in Section 5.3.4. With dynamic scoping, the variables that can be referred to by a function or a subroutine are determined by the sequence of calls made during execution, not by the nesting of blocks in the source program. It would be very difficult to compile such languages efficiently and allow for dynamic changes in the types of variables and the scope of names. These features can be more easily handled by an interpreter, which provides delayed binding of symbolic variable names to data types and locations.

Further discussions of the construction and use of interpreters can be found in Watt (1993).

5.4.3 P-Code Compilers

P-code compilers (also called *bytecode* compilers) are very similar in concept to interpreters. In both cases, the source program is analyzed and converted into an intermediate form, which is then executed interpretively. With a P-code compiler, however, this intermediate form is the machine language for a hypothetical

computer, often called a *pseudo-machine* or *P-machine*. The process of using such a P-code compiler is illustrated in Fig. 5.29. The source program is compiled, with the resulting object program being in P-code. This P-code program is then read and executed under the control of a P-code interpreter.

The main advantage of this approach is portability of software. It is not necessary for the compiler to generate different code for different computers, because the P-code object programs can be executed on any machine that has a P-code interpreter. Even the compiler itself can be transported if it is written in the language that it compiles. To accomplish this, the source version of the compiler is compiled into P-code; this P-code can then be interpreted on another computer. In this way, a P-code compiler can be used without modification on a wide variety of systems if a P-code interpreter is written for each different machine. Although writing such an interpreter is not a trivial task, it is certainly easier than writing a new compiler for each different machine. The same approach can also be used to transport other types of system software without rewriting.

The design of a P-machine and the associated P-code is often related to the requirements of the language being compiled. For example, the P-code for a Pascal compiler might include single P-instructions that perform array-subscript calculations, handle the details of procedure entry and exit, and perform elementary operations on sets. This simplifies the code-generation process, leading to a smaller and more efficient compiler. In addition, the P-code object program is often much smaller than a corresponding machine-code program would be. This is particularly useful on machines with severely limited memory size.

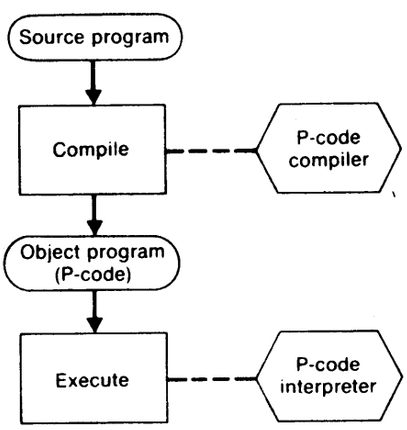


Figure 5.29 Translation and execution using a P-code compiler.

Obviously the interpretive execution of a P-code program may be much slower than the execution of the equivalent machine code. Depending upon the environment, however, this may not be a problem. Many P-code compilers are designed for a single user running on a dedicated microcomputer system. In that case, speed of execution may be relatively insignificant because the limiting factor in system performance may be the response time and "think time" of the user. If execution speed is important, some P-code compilers support the use of machine-language subroutines. By rewriting a small number of commonly used routines in machine language, rather than P-code, it is often possible to achieve substantial improvements in performance. Of course, this approach sacrifices some of the portability associated with the use of P-code compilers.

Section 5.5.2 of this text describes a recently developed P-code compiler for the Java language.

5.4.4 Compiler-Compilers

The process of writing a compiler usually involves a great deal of time and effort. In some areas, particularly the construction of scanners and parsers, it is possible to perform much of this work automatically. A *compiler-compiler* is a software tool that can be used to help in the task of compiler construction. Such tools are also often called *compiler generators* or *translator-writing systems*.

The process of using a typical compiler-compiler is illustrated in Fig. 5.30. The user (i.e., the compiler writer) provides a description of the language to be translated. This description may consist of a set of lexical rules for defining tokens and a grammar for the source language. Some compiler-compilers use this information to generate a scanner and a parser directly. Others create tables for use by standard table-driven scanning and parsing routines that are supplied by the compiler-compiler.

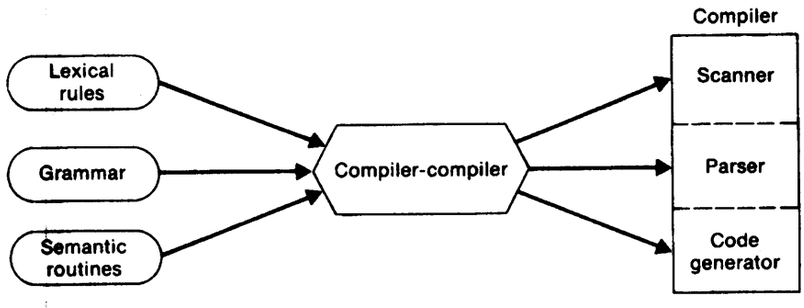


Figure 5.30 Automated compiler construction using a compiler-compiler.

In addition to the description of the source language, the user provides a set of semantic or code-generation routines. Often there is one such routine for each rule of the grammar, as we discussed in Section 5.1. This routine is called by the parser each time it recognizes the language construct described by the associated rule. However, some compiler-compilers can parse a larger section of the program before calling a semantic routine. In that case, an internal form of the statements that have been analyzed, such as a portion of the parse tree, may be passed to the semantic routine. This latter approach is often used when code optimization is to be performed. Compiler-compilers frequently provide special languages, notations, data structures, and other similar facilities that can be used in the writing of semantic routines.

The main advantage of using a compiler-compiler is, of course, ease of compiler construction and testing. The amount of work required from the user varies considerably from one compiler-compiler to another depending upon the degree of flexibility provided. Compilers that are generated in this way tend to require more memory and compile programs more slowly than hand-written compilers. However, the object code generated by the compiler may actually be better when a compiler-compiler is used. Because of the automatic construction of scanners and parsers, and the special tools provided for writing semantic routines, the compiler writer is freed from many of the mechanical details of compiler construction. The writer can therefore focus more attention on good code generation and optimization.

A brief description of one compiler-compiler (YACC) is given in Section 5.5.3. Further discussions and examples of compiler-writing tools can be found in Fischer and LeBlanc (1988).

5.5 IMPLEMENTATION EXAMPLES

In this section we briefly discuss the design of several real compilers. Section 5.5.1 describes the SunOS C compiler, which runs on a variety of hardware platforms.

Section 5.5.2 discusses the Java programming language and run-time environment recently developed by Sun Microsystems.

Section 5.5.3 presents a description of the YACC compiler-compiler, originally developed at Bell Laboratories for use with the UNIX operating system. We also briefly describe LEX, a scanner generator that is commonly used with YACC.

As in our previous discussions of real systems, we do not attempt to give a complete description of any of these compilers. References are provided for those readers who want more information.

5.5.1 SunOS C Compiler

The SunOS C compiler runs on a variety of hardware platforms, including SPARC, x86, and PowerPC. It conforms to the ANSI C standard, which is described in Schildt (1990). It also supports traditional (pre-ANSI) C features, as described in Kernighan and Ritchie (1978). The user of the compiler can specify which sets of language features are to be accepted by the compiler and which are to generate warning messages during compilation.

The translation process begins with the execution of the C preprocessor, which performs such functions as file inclusion and macro processing. (See Section 4.4.3 for a brief discussion of some of these features.) The output from the preprocessor goes to the C compiler itself. Several different levels of code optimization can be specified by the user. The compiler generates assembler language, which is then translated by an assembler. The preprocessor and compiler also accept source files that contain assembler language subprograms, and pass these on to the assembly phase.

The preprocessing phase consists of the following conceptual steps. The logical ordering of these steps is specified by the ANSI standard to eliminate possible ambiguities. The implementation of a particular preprocessor may in fact combine several of these steps. However, the effect must be the same as if they were executed separately in the sequence given.

1. Trigraph sequences are replaced by their single-character equivalents. The trigraph sequences are provided as a way to specify C language characters that may not available on some terminals. For example, the sequence ??< is replaced by {.
2. Any source line that ends with a backslash (\) and a newline is spliced together with the following line by deleting the backslash and newline.
3. The source file is partitioned into preprocessing tokens and sequences of white-space characters. Each comment is, in effect, replaced by one space. Preprocessing tokens include the regular tokens of the C language, plus tokens such as header file names and special numeric forms that are used only in the preprocessing stage.
4. Preprocessing directives are executed, and macros are expanded. Any source files that are included in response to an #include directive are processed from step 1 through 4.
5. Escape sequences in character constants and string literals are converted to their character equivalents. For example, \n is converted to a newline, and \0 is converted to the character with ASCII code 0.

6. Adjacent string literals are concatenated. For example, "hello," "world" is converted to "hello, world".

After preprocessing is complete, the actual process of program translation begins. The lexical analysis of the program is performed during preprocessing (step 3 above). Thus the compiler itself begins with syntactic analysis, followed by semantic analysis and code generation.

Like most implementations of UNIX, the SunOS operating system itself is largely written in C. Many compilers, editors, and other pieces of UNIX system software are also written in C. Because of this, it is important that a C compiler for such a system be able to generate efficient object code. It is also desirable that the compiler include tools to assist programmers in analyzing the performance of their programs. We will focus on these aspects of the SunOS C compiler.

Four different levels of code optimization can be specified by the user when a program is compiled. These levels are designated by O1 through O4. The O1 level does only a minimal amount of local (peephole) optimization. This type of optimization is performed at the assembler-language level, after the compilation itself is complete.

The O2 level provides basic local and global optimization. This includes register allocation and merging of basic blocks (see Section 5.2.2) as well as elimination of common subexpressions and removal of loop invariants (see Section 5.3.2). It also includes a number of other optimizations such as algebraic simplification and tail recursion elimination. In general, the O2 level of optimization results in the minimum object code size. This is the default that is provided unless otherwise requested by the user.

The O3 and O4 levels include optimizations that can improve execution speed, but usually produce a larger object program. For example, O3 optimization performs loop unrolling, which partially converts loops into straight-line code. The O4 level automatically converts calls to user-written functions into in-line code. This eliminates the overhead of calling and returning from the functions. Optimizations such as these can be guided by the user via compile-time options. For example, the user can specify the names of functions that should (or should not) be converted to in-line code.

When requested, the SunOS C compiler can insert special code into the object program to gather information about its execution. For example, one option accumulates a count of how many times each basic block is executed. Another option invokes a run-time recording mechanism. The resulting data can be analyzed by other SunOS software tools to produce profiles of program execution. For example, one such profile shows the percentage of execution time spent in different parts of the program.

Other program analysis tools provide support for reordering object code at the function level. To use these tools, the user instructs the C compiler to place

each function in a separate section. After the execution profile is analyzed, the object program is relinked to rearrange the functions. This can produce an executable program with improved locality of reference that runs more efficiently under a virtual memory management system. (Issues of virtual memory and locality of reference are discussed in Section 6.2.5.)

Like many other compilers, SunOS C can also generate information that supports the operation of debugging tools. For example, one option requests the compiler to include source code information in the object program. Using this information, the symbolic debugger can allow the user to examine variable values, control execution, browse the source file, and so on.

Further information about the SunOS C compiler can be found in Sun Microsystems (1994c).

5.5.2 Java Compiler and Environment

Java is a new programming language and operating environment developed by Sun Microsystems. It was designed to support applications in a diverse environment such as the Internet. Using Java, programmers can create high-performance applications that are portable without modification to multiple operating systems and hardware platforms. Java also provides features that are intended to make distributed applications more reliable and more secure.

The Java language itself is derived from C and C++. However, many of the features found in these languages have been removed to make Java programming as simple and error-free as possible. For example, Java has no "go to" statement and no pointers. Memory management is handled automatically, thus freeing the programmer from a complex and error-prone task. Data conversions that might cause loss of precision must be done explicitly by using built-in language features. This makes it possible for the compiler to perform strong type-checking, which leads to early detection of many programming errors.

Java is an object-oriented language. (If you are unfamiliar with the principles of object-oriented programming, you may want to review the discussion in Section 8.4.1.) The object-orientation in Java is stronger than in many other languages, such as C++. Except for a few primitive data types, everything in Java is an object. Arrays and strings are treated as objects. Even the primitive data types can be encapsulated inside objects if necessary. There are no procedures or functions in Java; classes and methods are used instead. Thus programmers are constrained to use a "pure" object-oriented style, rather than mixing the procedural and object-oriented approaches.

Java provides built-in support for multiple *threads* of execution. This feature allows different parts of an application's code to be executed concurrently.

For example, an interactive application might use one thread to run an animation, another to control sound effects, and a third to scroll a window. In Java, threads are implemented as objects. The Java library provides methods that can be invoked to start or stop a thread, check on the status of a thread, and synchronize the operation of multiple threads.

The Java compiler follows the P-code approach we discussed in Section 5.4.3. It does not generate machine code or assembly language for a particular target machine. Instead, the compiler generates *bytecodes*—a high-level, machine-independent code for a hypothetical machine (the Java Virtual Machine). This hypothetical machine is implemented on each target computer by an interpreter and run-time system. Thus a Java application can be run, without modification and without recompiling, on any computer for which a Java interpreter exists. The Java compiler itself is written in Java. Therefore the compiler can also run on any machine with a Java interpreter.

The bytecode approach also allows easy integration of Java applications into the World Wide Web. A segment of Java bytecode (often called an *applet*) can be included in an HTML page, in much the same way an image can be included. When a Java-compatible Web browser is used to view the page, the code for the applet is downloaded and executed by the browser. The applet can then perform animation, play sound, and generally interact with the user in real time.

The Java Virtual Machine supports a standard set of primitive data types: 1-, 2-, 4-, and 8-byte integers, single- and double-precision floating-point numbers, and 16-bit character codes. These data representations are independent of the architecture of the target machine. The interpreter is responsible for emulating these data types using the underlying hardware. Thus, for example, the floating-point formats and the big- or little-endian storage of integers on the target machine have no effect on an application program.

A bytecode instruction on the Java Virtual Machine consists of a 1-byte opcode followed by zero or more operands. Many opcodes require no explicit operands in the instruction; instead, they take their operand values from a stack. A stack organization was chosen so that it would be easy to emulate the machine on a computer with few general-purpose registers (such as the x86 architecture).

For example, the "iadd" instruction adds two integers together. It expects that the integers to be added are the top two words on the operand stack, pushed there by previous instructions. Both integers are popped from the stack, and their sum is pushed back onto the stack. Each primitive data type has specialized instructions that must be used to operate on items of that type.

There are also single bytecode instructions that perform higher-level operations. For example, one instruction allocates a new array of a particular type. Other instructions can be used to transfer elements of an array to or from the operand stack.

Similarly, the bytecode instructions provide direct support for the object-oriented nature of Java. One instruction creates a new object of a specified type. Another instruction tests whether an object is an instance of a particular class. There are four instructions that can be used (depending upon the situation) to invoke a method on an object. Another group of instructions is used to manipulate fields within an object.

Performance is always a consideration, especially with interpreted execution. The Java interpreter is designed to run as fast as possible, without needing to check the run-time environment. The automatic *garbage collection* system used to manage memory runs as a low-priority background thread. Experiments conducted on modern (1995) systems such as workstations and high-end personal computers show generally good response running interactive graphical applications. For example, creating a new object typically requires about 8 msec and invoking a method on an object requires about 2 msec.

However, there are times when higher performance may be needed. In such cases, the Java bytecodes can be translated at execution time into machine code for the computer on which the application is running. Measurements on execution of bytecodes converted to machine code show performance roughly the same as the equivalent application coded directly in C or C++.

A description of the Java language can be found in Arnold and Gosling (1996). Further information about the implementation of Java can be found in Lindholm and Yellin (1996) and Sun Microsystems (1995b).

5.5.3 The YACC Compiler-Compiler*

YACC (Yet Another Compiler-Compiler) is a parser generator that is available on UNIX systems. YACC has been used in the production of compilers for Pascal, RATFOR, APL, C, and many other programming languages. It has also been used for several less conventional applications, including a typesetting language and a document retrieval system. In this section we give brief descriptions of YACC and LEX, the scanner generator that is related to YACC. Further information about these software tools can be found in Levine et al. (1992).

A lexical scanner must be supplied for use with YACC. This scanner is called by the parser whenever a new input token is needed. It returns an integer that identifies the type of token found, as described in Section 5.1. The scanner may also make entries in a symbol table for the identifiers that are processed.

LEX is a scanner generator that can be used to create scanners of the type required by YACC. A portion of an input specification for LEX is shown in

*Adapted from "Language Development Tools on the Unix System" by S.C. Johnson, from the IEEE publication *Computer*, Vol. 13, No. 8, pp. 16–21, August 1980. © 1980 IEEE.

Fig. 5.31(a). Each entry in the left-hand column is a pattern to be matched against the input stream. When a pattern is matched, the corresponding action routine in the right-hand column is invoked. These action routines are written in the programming language C. The routines usually return an indication of the token that was recognized. They may also make entries in tables and perform other similar tasks.

In the example shown in Fig. 5.31(a), the first pattern has no associated action; the effect of this is to delete blanks as the input is scanned. The actions for the next three patterns simply return a token type: the token LET for the keyword *let*, MUL for the operator *, and ASSIGN for the operator =. As discussed above, the internal representations of LET, MUL, and the other tokens are integers. The fifth pattern specifies the form of identifiers to be recognized. The first character must be in the range a–z or A–Z. This may be followed by any number of characters in the ranges a–z, A–Z, or 0–9. The * in this pattern indicates that an arbitrary number of repetitions of the preceding

```
.
.
.
" "                          ; /* ignore blanks */
let                          return(LET);
"*"                          return(MUL);
"="                          return(ASSIGN);
[a-zA-Z] [a-zA-Z0-9]*        {make entries in tables; return(ID)};
.
.
.
```

(a)

```
%token ASSIGN ID LET MUL ...
.
.
.
statement   :   LET  ID  ASSIGN expr
                     { ... }
expr        :   expr  MUL  expr
                     { $$ = build(MUL,$1,$3);}
expr        :   ID
                     { ... }
.                          .
.
.
```

(b)

Figure 5.31 Example of input specifications for LEX and YACC.

item are allowed. The action routine for this pattern makes entries in the appropriate tables to describe the identifier found and then returns the token type ID.

According to the specifications given in Fig. 5.31(a), the input

```
let x = y * z
```

would be scanned as the sequence of tokens

```
LET ID ASSIGN ID MUL ID
```

Note that the first pattern that matches the input stream is selected, so the keyword *let* is recognized as the token LET, not as ID.

LEX can be used to produce quite complicated scanners. Some languages, such as FORTRAN, however, have lexical analyzers that must still be generated or modified by hand.

The YACC parser generator accepts as input a grammar for the language being compiled and a set of actions corresponding to rules of the grammar. A portion of such an input specification appears in Fig. 5.31(b). The first line shown is a declaration of the token types used. The other entries are rules of the grammar. The YACC parser calls the semantic routine associated with each rule as the corresponding language construct is recognized. Each routine may return a value by assigning it to the variable $$. Values returned by previous routines, or by the scanner, may be referred to as $1, $2, etc. These variables designate the values returned for the components on the right-hand side of the corresponding rule, reading from left to right.

An example of the use of such values is shown in Fig. 5.31(b). The semantic routine associated with the rule

```
expr : expr MUL expr
```

constructs a portion of the parse tree for the statement, using a tree-building function *build*. This subtree is returned from the semantic routine by assigning the subtree to $$. The arguments passed to the *build* function are the operator MUL and the values (i.e., the subtrees) returned when the operands were recognized. These values are denoted by $1 and $3.

It is sometimes useful to perform semantic processing as each part of a rule is recognized. YACC permits this by allowing semantic routines to be written in the middle of a rule as well as at the end. The value returned by such a routine is available to any of the routines that appear later in the rule. It is also possible for the user to define global variables that can be used by all of the semantic routines and by the lexical scanner.

The parsers generated by YACC use a bottom-up parsing method called LALR(1), which is a slightly restricted form of shift-reduce parsing. The parsers produced by YACC have very good error detection properties. Error handling permits the reentry of the items in error or a continuation of the input process after the erroneous entries are skipped.

EXERCISES

Section 5.1

1. Draw parse trees, according to the grammar in Fig. 5.2, for the following <id-list>s:
   a. `ALPHA`
   b. `ALPHA, BETA, GAMMA`

2. Draw parse trees, according to the grammar in Fig. 5.2, for the following <exp>s:
   a. `ALPHA + BETA`
   b. `ALPHA - BETA * GAMMA`
   c. `ALPHA DIV (BETA + GAMMA) - DELTA`

3. Modify the grammar in Fig. 5.2 to include statements of the form

   ```
   IF condition THEN statement-1 ELSE statement-2
   ```

   where the ELSE clause may be omitted. Assume that the condition must be of the form $a < b$, $a = b$, or $a > b$, where a and b are single identifiers or integers. You do not need to allow for nested IFs—that is, *statement-1* and *statement-2* may not be IF statements.

4. Write BNF grammar for JAVA.

5. Modify the grammar in Fig. 5.2 so that the I/O list for a WRITE statement may include character strings enclosed in quotation marks, as well as identifiers.

6. Write an algorithm that scans an input stream, recognizing operators and identifiers. An identifier may be up to 10 characters long. It must start with a letter, and the remaining characters, if any, must be letters and digits. The operators to be recognized are +, –, *, DIV,

and :=. Your algorithm should return an integer that represents the type of token found, using the coding scheme of Fig. 5.5. If an illegal combination of characters is found, the algorithm should return the value –1.

7. Modify the scanner you wrote in Exercise 8 so that it recognizes integers as well as identifiers. Integers may begin with a sign (+ or –); however, they may not begin with the digit 0 (except for the integer that consists of a single 0).

8. Draw a state diagram for a finite automaton to recognize a token type named "real constant." This token consists of a string of digits that contains a decimal point. There must be at least one digit before the decimal point.

9. Modify your answer to Exercise 10 so that a real constant may also contain a scale factor. The scale factor, which follows the string of digits, consists of the letter E followed by a positive or negative integer. A real constant must contain either a decimal point or a scale factor (or both). There must be at least one digit before the decimal point (if any).

10. Draw a state diagram for a finite automaton to recognize a token type named "write-element." Each such token must have one of the following forms:

```
name
name:n
name:n:m
'string'
'string':n
```

where

name must start with a letter (a–z); all characters after the first letter must be either letters (a–z) or digits (0–9).

string may contain any characters other than quote (').

n,m must be positive integers containing only digits (0–9), with no leading zeros allowed.

11. Write a program that simulates the operation of a finite automaton, using a tabular representation like the one illustrated in Fig. 5.10(b).

12. Select a high-level programming language with which you are familiar and write a lexical scanner for it.

13. Parse the assignment statement on line 11 of Fig. 5.1, using the method of recursive descent and the procedures given in Fig. 5.13.

14. Write recursive-descent parsing procedures that correspond to the rules for <dec-list>, <dec>, and <type> in Fig. 5.11. Use these procedures to parse the declaration on line 3 of Fig. 5.1.

15. Write recursive-descent parsing procedures for the remaining nonterminals in the grammar of Fig. 5.11. Parse the entire program in Fig. 5.1, using the method of recursive descent.

16. Use the routines in Figs. 5.14–5.16 to generate code for the following statements from the example program in Fig. 5.1:
    a. the assignment statement on line 11
    b. the WRITE statement on line 15
    c. the FOR statement beginning on line 7

    Refer to the parse tree in Fig. 5.4 to see the order in which the parser recognizes the various constructs involved in these statements.

17. Use the routines in Figs. 5.14–5.16 to generate code for the entire program in Fig. 5.1.

18. Write code-generation routines for the new rules that you added to the grammar in Exercise 6 to define the IF statement.

19. Suppose that the grammar in Fig. 5.2 is modified to allow floating-point variables (i.e., the <type> REAL) as well as integers. How would the code-generation routines given in the text need to be changed? Assume that mixed-mode arithmetic expressions are allowed according to the usual rules of Pascal.

20. The code-generation routines in the text use immediate addressing for integers written by the programmer in arithmetic expressions (for example, the 100 in the expression SUM DIV 100). How could such constants be handled by a compiler for a machine that does not have immediate addressing?

21. What kinds of source program errors would be detected during lexical analysis?

22. What kinds of source program errors would be detected during syntactic analysis?

23. What kinds of source program errors would be detected during code generation?

24. In what ways might the symbol table used by a compiler be different from the symbol table used by an assembler?

25. Suppose you have a one-pass Pascal compiler similar to the one described in Section 5.1. Now you want to add a simple macro capability to this compiler. The macro processing should be integrated into the rest of the compiler, not implemented as a preprocessor. Describe how the macro processing routines would interact with the rest of the compiler. For example, would the routine that processes macro definitions be called by the scanner, the parser, or the code generator? Which of these phases of the compiler would interact with the routines that recognize and expand macro invocation statements?

Section 5.2

1. Rewrite the code-generation routines given in Figs. 5.14 and 5.15 to produce quadruples instead of object code.

2. Write a set of routines to generate object code from the quadruples produced by your routines in Exercise 1. (Hint: You will need a routine that is similar in function to the GETA procedure in Fig. 5.15.)

3. Use the routines you wrote in Exercise 1 to produce quadruples for the following program fragment:

```
READ(X,Y);
Z := 3 * X - 5 * Y + X * Y;
```

4. Use the routines you wrote in Exercise 2 to produce object code from the quadruples generated in Exercise 3.

5. Rewrite the code-generation routines given in Fig. 5.16 to produce quadruples instead of object code.

6. Use the routines you wrote in Exercises 1 and 5 to produce quadruples for the program in Fig. 5.1.

7. Divide the quadruples you produced in Exercise 6 into basic blocks and draw a flow graph for the program.

8. Assume that you are generating SIC/XE object code from the quadruples produced in Exercise 6. Show one way of performing

register assignments to optimize the object code, using registers S and T to hold variable values and intermediate results.

Section 5.3

1. Write an algorithm for the prologue of a procedure, assuming the activation record format shown in Fig. 5.26.

2. Write an algorithm for the epilogue of a procedure, assuming the activation record format shown in Fig. 5.26.

3. Suggest a way of using the activation record stack to perform dynamic storage allocation for controlled variables. What would be the advantages and disadvantages of such a technique as compared to using a separate area of free storage to perform these allocations?

4. Write algorithm for scanner, parser and code generator for input/output, definitions, control statements of C language.

5. Assume the array C is declared as

   ```
   C: ARRAY[5..20] OF INTEGER
   ```

   Generate quadruples for the statement

   ```
   C[I] := 0
   ```

6. Assume the array D is declared as

   ```
   D: ARRAY [-10..10,2..12] OF INTEGER
   ```

   and is stored in *row-major* order. Generate quadruples for the statement

   ```
   D[I,J] := 0
   ```

7. Assume the array D declared in Exercise 5 is stored in *column-major* order. Generate quadruples for the statement

   ```
   D[I,J] := 0
   ```

8. Generalize the methods given in Section 5.3.1 to the storage of three-dimensional arrays in row-major order. Assuming the array declaration

   ```
   E : ARRAY[1..5, 1..10, 0..8] OF INTEGER
   ```

   generate quadruples for the statement

   ```
   E[I,J,K] := 0
   ```

9. How could the base address for the array A defined in Fig. 5.22(a) be modified to avoid the need for subtracting 1 from the subscript value (quadruple 1)?

10. How could the technique derived in Exercise 8 be extended to two-dimensional arrays?

11. Assume the array declaration

```
T : ARRAY[1..5, 1..100] of INTEGER
```

Translate the following statements into quadruples and perform elimination of common subexpressions on the result.

```
K := J-1;
FOR I := 1 TO 5 DO
   BEGIN
      T[I,J] := K * K;
      J := J + K;
      T [I,J] := K * K - 1;
   END
```

12. Modify the quadruples produced in Exercise 10 to remove loop invariants.

13. Write an algorithm to construct the proper display when a procedure is invoked. Your algorithm may use the old display (i.e., the current display before the call), the address of the activation record created for the procedure being called, and the block-nesting level of the procedure being called.

Chapter 6

Operating Systems

In this chapter we discuss the functions and design of operating systems. We discuss the most important concepts and issues related to operating systems, giving examples and providing references for further reading.

Operating systems vary widely in purpose and design. Some are relatively simple systems designed to support a single user on a personal computer. Others are extremely complex systems that support many concurrent users and manage highly sophisticated hardware and software resources. Section 6.1 discusses the basic features of an operating system that should be found in almost any such piece of software. Because of the large variety of operating systems, this list of basic features is surprisingly brief. It consists of only a few generic functions that could almost be taken as a definition of the term *operating system.*

Section 6.2 describes some important machine-dependent operating system features. Section 6.3 describes a number of machine-independent features. Many of the functions discussed in these two sections are actually required in operating systems that support more than one user at a time. Such functions include, for example, the allocation of system resources and the management of communication among different users.

Section 6.4 briefly presents some design alternatives for operating systems. Section 6.5 describes a number of actual operating systems, illustrating some of the variety of form and function in such software.

6.1 BASIC OPERATING SYSTEM FUNCTIONS

In this section we discuss the fundamental functions common to all operating systems. The main purpose of an operating system is to make the computer easier to use. That is, the software provides an interface that is more user-friendly than the underlying hardware. As a part of this process, the operating system manages the resources of the computer in an attempt to meet overall system goals such as efficiency. The details of this resource management can be quite complicated; however, the operating system usually hides such complexities from the user.

The basic functions of an operating system is shown in Fig. 6.1. The operating system provides programs with a set of services that can aid in the performance of many common tasks. For example, suppose program P wants to read data sequentially from a file. An operating system might provide a service routine that could be invoked with a command such as *read(f)*. With such a command, the program would specify a file name; the operating system would take care of the details of performing the actual machine-level I/O.

The most common ways of classifying operating systems are based on the kind of user interface provided. Much operating system terminology arises from the way the system appears to a user. In this section we introduce terms commonly used to describe operating systems. The types of systems mentioned are not always distinct. Some of the classifications overlap and many real operating systems fall into more than one category.

One way of classifying operating systems is concerned with the number of users the system can support at one time. A *single-job* system is one that runs one user job at a time. Single-job systems, which are commonly found today on microcomputers and personal computers, were the earliest type of operating system. A single-job operating system would probably be used on a standard SIC computer. Because of the limited memory size and lack of data channels and other resources, it would be difficult to support more than one user on such a machine.

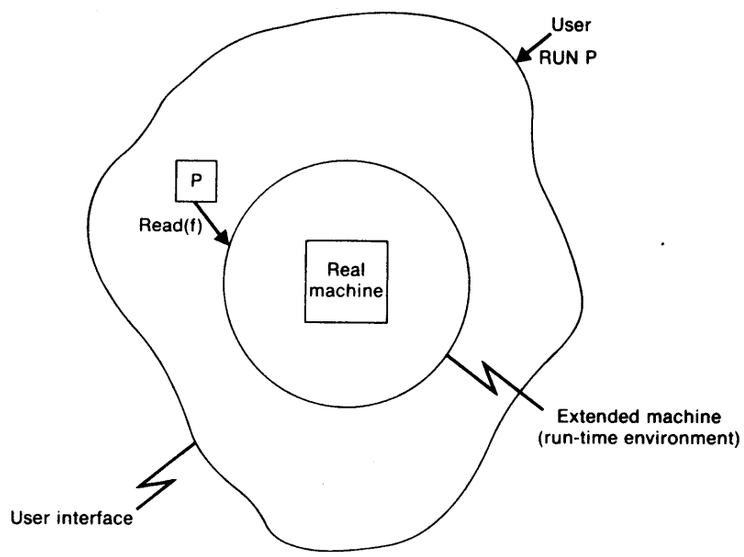


Figure 6.1 Basic concept of an operating system.

A *multiprogramming* system permits several user jobs to be executed concurrently. The operating system takes care of switching the CPU among the various user jobs. It also provides a suitable run-time environment and other support functions so the jobs do not interfere with each other. A *multiprocessor* system is similar to a multiprogramming system, except that there is more than one CPU available. In most multiprocessor systems, the processors share a common memory. Thus the user can view the system as if it were a powerful single processor.

A network of computers may be organized in a number of different ways. Each computer may have its own independent operating system, which provides an interface to allow communication via the network. A user of such a system is aware of the existence of the network. He or she may login to remote machines, copy files from one machine to another, etc. This kind of system is often called a *network operating system.* Except for the network interface, such an operating system is quite similar to those found on a single-computer system.

A *distributed operating system* allows a more complex type of network organization. This kind of operating system manages hardware and software resources so that a user views the entire network as a single system. The user is unaware of which machine on the network is actually running a program or storing data. (In fact, many such systems allow programs to run on several processors at the same time.)

Historically, operating systems have also been classified by the type of access provided to a user. In a *batch processing* system, a job is described by a sequence of control statements stored in a machine-readable form. The operating system can read and execute a series of such jobs without human intervention except for such functions as tape and disk mounting. The order in which the jobs are executed can be selected in several different ways. This job scheduling problem is discussed in Section 6.3.2. A *time-sharing* system provides interactive, or conversational, access to a number of users. The operating system executes commands as they are entered, attempting to provide each user with a reasonably short response time to each command. A *real-time* system is designed to respond quickly to external signals such as those generated by data sensors. Real-time systems are used, for example, on computers that monitor and control time-critical processes such as nuclear reactor operation or spacecraft flight.

In general, the goal of a multiprogramming batch processing system is to make the most efficient use of the computer. On the other hand, the goal of a time-sharing system is to provide good response time to the interactive users. To provide good response time, it may be necessary to accept less efficient machine utilization. The goal of a real-time system is to provide a *guaranteed* response time to time-critical external events. It is quite common for these goals to be mixed in a single operating system. For example, many batch

processing systems also support time-sharing users, and some may also provide support for real-time applications.

Further discussions concerning these types of operating systems and descriptions of their historical background can be found in Tanenbaum (1992) and Singhal and Shivaratri (1994).

6.2 MACHINE-DEPENDENT OPERATING SYSTEM FEATURES

One of the most important functions of an operating system is managing the resources of the computer on which it runs. Many of these resources are directly related to hardware units such as central memory, I/O channels, and the CPU. Thus many operating system functions are closely related to the machine architecture.

Consider, for example, a standard SIC computer. This machine has a small central memory, no I/O channels, no supervisor-call instruction, and no interrupts. Such a machine might be suitable as a personal computer for a single user; however, it could not reasonably be shared among several concurrent users. Thus an operating system for a standard SIC machine would probably be a single-job system, providing a simple user interface and a minimal set of functions in the run-time environment. It would probably provide few, if any, capabilities beyond the simple ones discussed in Section 6.1.

On the other hand, a SIC/XE computer has a larger central memory, I/O channels, and many other hardware features not present on the standard SIC machine. A computer with these characteristics might well have a multiprogramming operating system. Such a system would allow several concurrent users to share the expanded machine resources that are available, and would take better advantage of the more advanced hardware. Of course, the sharing of a computing system between several users creates many problems, such as resource allocation, that must be solved by the operating system. In addition, the operating system must provide support for the more advanced hardware features such as I/O channels and interrupts.

In this section we discuss some important machine-dependent operating system functions. This discussion is presented in terms of a SIC/XE computer; however, the same principles can easily be applied to other machines that have architectural features similar to those of SIC/XE. We describe a number of significant SIC/XE hardware features as a part of our discussion. For ease of reference, these features are also summarized in Appendix C.

Section 6.2.1 introduces fundamental concepts of interrupts and interrupt processing that are used throughout the remainder of this chapter. Section 6.2.2 discusses the problem of switching the CPU among the several user jobs

being multiprogrammed. Section 6.2.3 describes a method for managing input and output using I/O channels in a multiprogramming operating system. Sections 6.2.4 and 6.2.5 discuss the problem of dividing the central memory between user jobs. Section 6.2.4 presents techniques for managing real memory, and Section 6.2.5 introduces the important topic of virtual memory.

6.2.1 Interrupt Processing

An *interrupt* is a signal that causes a computer to alter its normal flow of instruction execution. Such signals can be generated by many different conditions, such as the completion of an I/O operation, the expiration of a preset time interval, or an attempt to divide by zero.

The sequence of events that occurs in response to an interrupt is illustrated in Fig. 6.2. Suppose program A is being executed when an interrupt signal is generated by some source. The interrupt automatically transfers control to an *interrupt-processing routine* (also called an *interrupt handler*) that is usually a part of the operating system. This interrupt-processing routine is designed to take some action in response to the condition that caused the interrupt. After completion of the interrupt processing, control can be returned to program A at the point at which its execution was interrupted.

In the sequence of events just described, the generation and processing of the interrupt may be completely unrelated to program A. For example, the interrupt might be generated by the completion of an I/O operation requested by

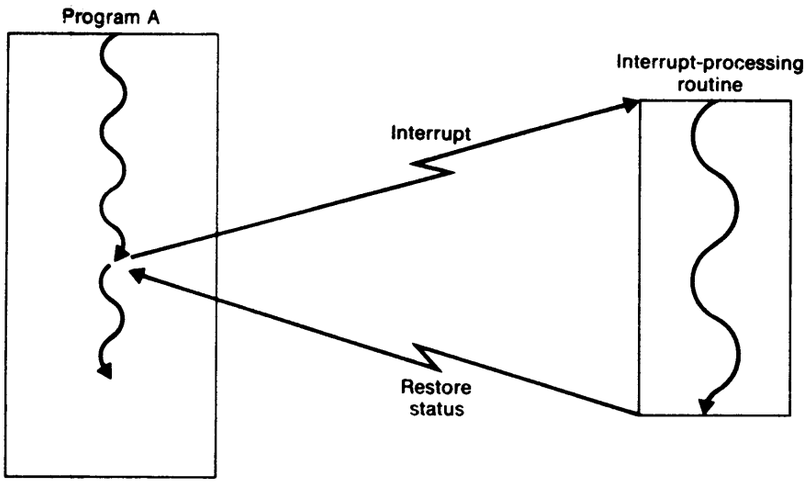


Figure 6.2 Basic concept of interrupt processing.

another program. In general, it is impossible to predict when, and for what reason, program A will be interrupted in this way. Another way of expressing this is to say that the interrupts may be *asynchronous* with respect to program A. The hardware and software take care of saving the status of the computer when A is interrupted, and restoring it when A is resumed. Because of this, the execution of A is unaffected, except for timing, by the occurrence of the interrupt. Indeed, there may be no direct way for A even to detect that an interrupt has occurred.

Figure 6.3 describes the four classes of interrupts on a SIC/XE computer. An *SVC interrupt* (Class I) is generated when a supervisor call instruction is executed by the CPU. This instruction is used by programs to request operating system functions. A *program interrupt* (Class II) is generated by some condition that occurs during program execution, such as an attempt to divide by zero or an attempt to execute an illegal machine instruction. Appendix C contains a complete list of the conditions that can cause a program interrupt.

A *timer interrupt* (Class III) is generated by an interval timer within the CPU. This timer contains a register that can be set to an initial positive value by the privileged instruction STI. The value in this register is automatically decremented by 1 for each millisecond of CPU time that is used. When the value reaches zero, a timer interrupt occurs. The interval timer is used by the operating system to govern how long a user program can remain in control of the machine.

An *I/O interrupt* (Class IV) is generated by an I/O channel or device. Most such interrupts are caused by the normal completion of some I/O operation; however, an I/O interrupt may also signal a variety of error conditions.

When an interrupt occurs, the status of the CPU is saved, and control is transferred to an interrupt-processing routine. We describe the method used by SIC/XE to accomplish this. The mechanism described is typical of the ones used on many real computers.

On a SIC/XE machine, there is a fixed *interrupt work area* corresponding to each class of interrupt, as illustrated in Fig. 6.4. For example, the area assigned

Class	Interrupt type
I	SVC
II	Program
III	Timer
IV	I/O

Figure 6.3 SIC/XE interrupt types.

to the timer interrupt begins at memory address 160. When a timer interrupt occurs, the contents of all registers are stored in this work area, as shown in Fig. 6.4(a). Then the status word SW and the program counter PC are loaded with values that are prestored in the first two words of the area. This storing and loading of registers is done automatically by the hardware of the machine.

The loading of the program counter PC with a new value automatically causes a transfer of control. The next instruction to be executed is taken from the address given by the new value of PC. This address, which is prestored in the interrupt work area, is the starting address of the interrupt-handling routine for a timer interrupt. The loading of the status word SW also causes certain changes, described later in this section, in the state of the CPU.

After taking whatever action is required in response to the interrupt, the interrupt-handling routine returns control to the interrupted program by executing a Load Processor Status (LPS) instruction. This action is illustrated in Fig. 6.4(b). LPS causes the stored contents of SW, PC, and the other registers to be loaded from consecutive words beginning at the address specified in the instruction. This restores the CPU status and register contents that existed at the time of the interrupt, and transfers control to the instruction following the one that was being executed when the interrupt occurred. The saving and restoring of the CPU status and register contents are often called *context switching* operations.

The status word SW contains several pieces of information that are important in the handling of interrupts. We discuss the contents of SW for a SIC/XE machine. Most computers have a similar register, which is often called a *program status word* or a *processor status word.*

Figure 6.5 shows the contents of the status word SW. The first bit, MODE, specifies whether the CPU is in user mode or supervisor mode. Ordinary programs are executed in user mode (MODE = 0). When an interrupt occurs, the new SW contents that are loaded have MODE = 1, which automatically switches the CPU to supervisor mode so that privileged instructions may be used. Before the old value of SW is saved, the ICODE field is automatically set to a value that indicates the cause of the interrupt. For an SVC interrupt, ICODE is set to the value supplied by the user in the SVC instruction. This value specifies the type of service request being made. For a program interrupt, ICODE indicates the type of condition, such as division by zero, that caused the interrupt. For an I/O interrupt, ICODE gives the number of the I/O channel that generated the interrupt. Further information about the possible values of ICODE can be found in Appendix C.

The status word also contains the condition code CC. Saving SW automatically preserves the condition code value that was being used by the interrupted process. The use of the fields IDLE and ID will be described later in this chapter. IDLE specifies whether the CPU is executing instructions or

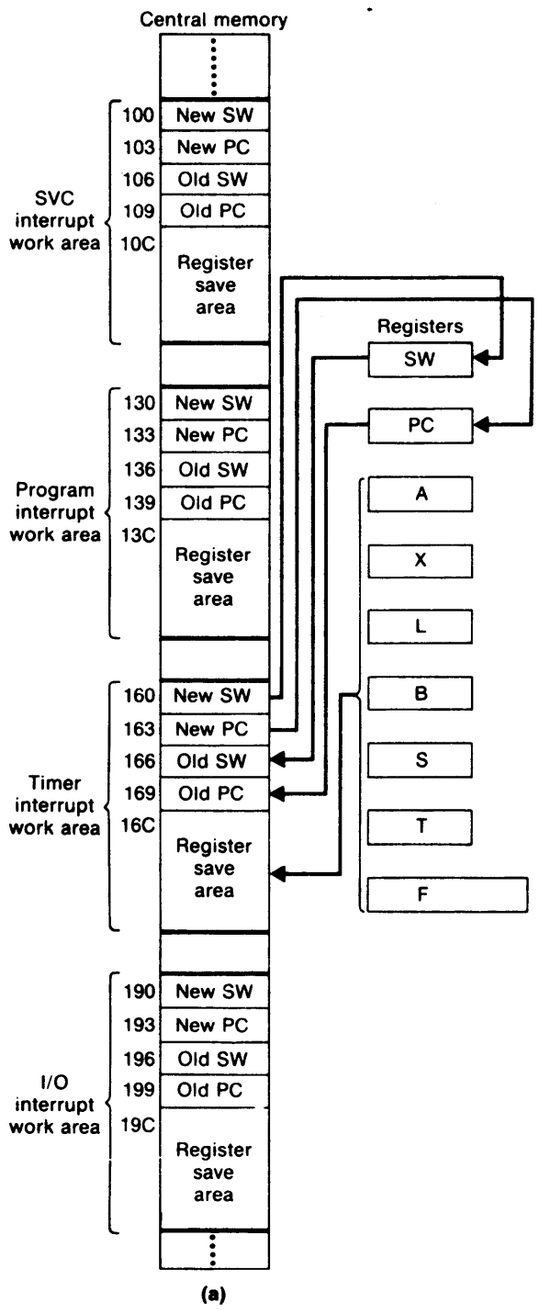


Figure 6.4 Context switching operations caused by (a) timer interrupt and (b) LPS 166.

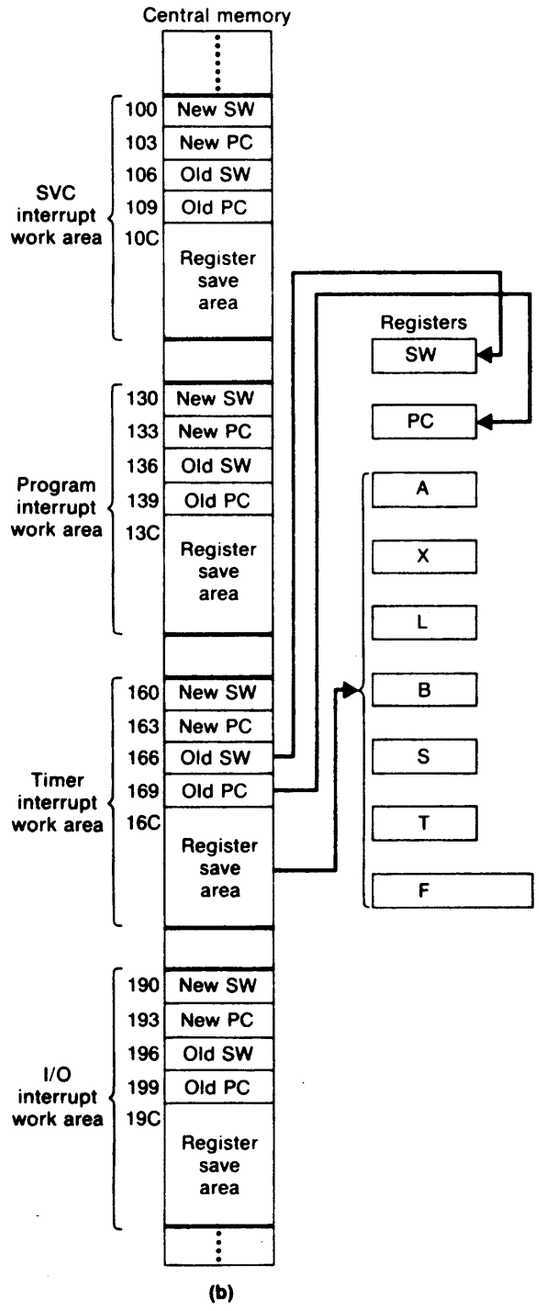


Figure 6.4 (*cont'd*)

Bit position	Field name	Use
0	MODE	0 = user mode, 1 = supervisor mode
1	IDLE	0 = running, 1 = idle
2–5	ID	Process identifier
6–7	CC	Condition code
8–11	MASK	Interrupt mask
12–15		Unused
16–23	ICODE	Interruption code

Figure 6.5 SIC/XE status word contents.

is idle. ID contains a 4-bit value that identifies the user program currently being executed.

The remaining status word field, MASK, is used to control whether interrupts are allowed. This control is necessary to prevent loss of the stored processor status information. Suppose, for example, that an I/O interrupt occurs. The values of SW, PC, and the other registers would be stored in the I/O-interrupt work area as just described, and the CPU would begin to execute the I/O-interrupt handler. If another I/O interrupt occurred before the processing of the first had been completed, another context switch would take place. This time, however, the register contents stored in the interrupt work area would be the values currently being used by the interrupt handler. The values that were saved by the original interrupt would be destroyed, so it would be impossible to return control to the user program that was executing at the time of the first interrupt.

To avoid such a problem, it is necessary to prevent certain interrupts from occurring while the first one is being processed. This is accomplished by using the MASK field in the status word. MASK contains one bit that corresponds to each class of interrupt. If a bit in MASK is set to 1, interrupts of the corresponding class are allowed to occur. If the bit is set to 0, interrupts of the corresponding class are not allowed. When interrupts are prohibited, they are said to be *masked* (also often called *inhibited* or *disabled*). Interrupts that are masked are not lost, however, because the hardware saves the signal that would have caused the interrupt. An interrupt that is being temporarily delayed in this way is said to be *pending*. When interrupts of the appropriate class are again permitted, because MASK has been reset, the signal is recognized and an interrupt occurs.

The masking of interrupts on a SIC/XE machine is under the control of the operating system. It depends upon the value of MASK in the SW that is prestored in each interrupt work area. One approach is to set all the bits in MASK to 0, which prevents the occurrence of any other interrupt. However, it is not really necessary to inhibit all interrupts in this way.

Each class of interrupt on a SIC/XE machine is assigned an *interrupt priority*. SVC interrupts (Class I) have the highest priority, program interrupts (Class II) have the next highest priority, and so on. The MASK field in the status word corresponding to each interrupt class is set so that all interrupts of equal or lower priority are inhibited; however, interrupts of higher priority are allowed to occur. For example, the status word that is loaded in response to a program interrupt would have the MASK bits for program, timer, and I/O interrupts set to 0; these classes of interrupt would be inhibited. The MASK bit for SVC interrupts would be set to 1, so these interrupts would be allowed. When interrupts are enabled at the end of an interrupt-handling routine, there may be more than one type of interrupt pending (for example, one timer interrupt and one I/O interrupt). In such a case, the pending interrupt with the highest priority is recognized first.

With this type of priority scheme, it is possible for an interrupt-processing routine itself to be interrupted. Such a *nested* interrupt situation is illustrated in Fig. 6.6, which shows program A in control of the CPU when an I/O interrupt occurs. The status of A is then saved, and control passes to the I/O-interrupt handler. During the execution of this routine, a timer interrupt occurs, and

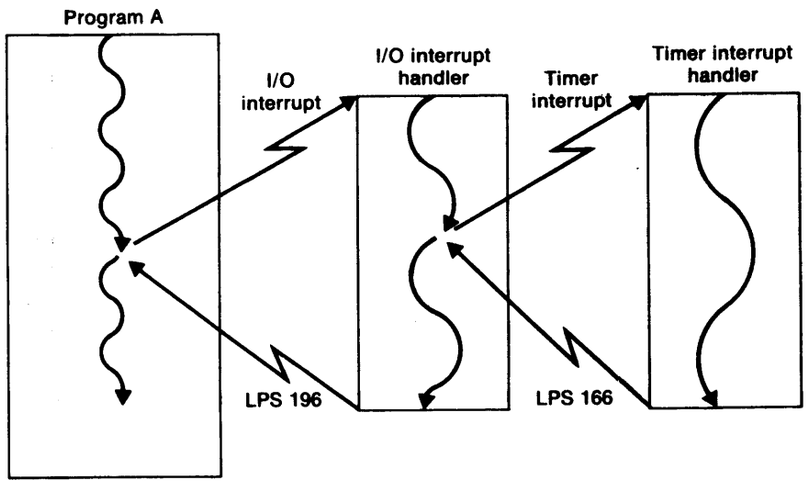


Figure 6.6 Example of nested interrupt processing.

control is transferred to the timer-interrupt handler. After the processing of the timer interrupt is completed, an LPS instruction is used to reload the processor status from the timer-interrupt work area. This returns control to the I/O-interrupt handler. Because the previous value of MASK is reloaded, timer interrupts, which had been inhibited, are once again allowed. I/O interrupts, however, are still inhibited. After the I/O-interrupt processing is completed, another LPS returns control to program A, restoring the CPU status as it was at the time of the original interrupt. At this time, all interrupts are allowed because the status word being used by program A has all MASK bits set to 1.

In later sections of this chapter, we see how interrupts can be used in such operating system functions as process scheduling, I/O management, and memory allocation.

6.2.2 Process Scheduling

A *process*, sometimes called a *task*, is most often defined as a program in execution. The CPU is assigned to processes by the operating system in order to perform computing work. In a single-job operating system, there is only one user process at a time. In a multiprogramming system, however, there may be many independent processes competing for control of the CPU. *Process scheduling* is the management of the CPU by switching control among the various competing processes according to some scheduling policy. The same techniques can also be applied to scheduling each individual CPU in a multiprocessor system.

In most cases, a process corresponds to a user job. However, some operating systems allow one user job to create several different processes that are executed concurrently. In addition, some systems allow one program to be executed by several independent processes. Further information about such topics can be found in Tanenbaum (1992). In our discussions we assume that each process corresponds to exactly one program and one user job.

A process is created when a user job begins execution, and this process is destroyed when the job terminates. During the period of its existence, the process can be considered to be in one of three states. A process is *running* when it is actually executing instructions using the CPU. A process is *blocked* if it must wait for some event to occur before it can continue execution. For example, a process might be blocked because it must wait for the completion of an I/O operation before proceeding. Processes that are neither blocked nor running are said to be *ready*. These processes are candidates to be assigned the CPU when the currently running process gives up control.

Figure 6.7 shows the possible transitions between these three process states. At any particular time, there can be no more than one process in the

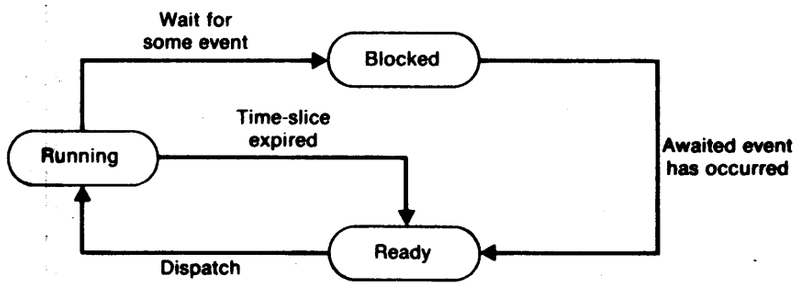


Figure 6.7 Process state transitions.

running state (i.e., in control of the CPU). When the operating system transfers control to a user process, it sets the interval timer to specify a *time-slice*, which is a maximum amount of CPU time the process is allowed to use before giving up control. If this time expires, the process is removed from the running state and placed in the ready state. The operating system then selects some process from the ready state, according to its scheduling policy. This process is placed in the running state and given control of the CPU. The selection of a process, and the transfer of control to it, is usually called *dispatching*. The part of the operating system that performs this function is known as the *dispatcher*.

Before it has used all its assigned time-slice, a running process may find that it must wait for the occurrence of some event such as the completion of an I/O operation. In such a case, the running process enters the blocked state, and a new process is dispatched. When an awaited event occurs, the blocked process associated with that event is moved to the ready state, where it is again a candidate for dispatching. The operations of waiting for an event, and of signaling that an event has occurred, are implemented as operating system service requests (using SVC). A mechanism often used to associate processes with awaited events is described later in this section.

A process is usually switched between the running, ready, and blocked states many times before completing its execution. Each time a process leaves the running state, its current status must be saved. This status must be restored the next time the process is dispatched so that the switching will have no effect on the results of the computation being performed. The status information for each process is saved by the operating system in a *process status block* (PSB) for that process. A PSB is created when a process first begins execution and is deleted when that process terminates. The PSB contains an indication of the process state (running, ready, or blocked), an area that is used to save all machine registers (including SW and PC), and a variety of other information (for example, an indication of the system resources used by the process).

whether or not the associated event has occurred. The ESB also contains a pointer to ESBQUEUE, a list of all processes currently waiting for the event. Further information about ESBs, and examples of their creation and use, are presented in Section 6.2.3.

The WAIT request is issued by a running process and indicates that the process cannot proceed until the event associated with ESB has occurred. Thus the algorithm for WAIT first examines ESBFLAG. If the event has already occurred, control is immediately returned to the requesting process. If the event has not yet occurred, the running process is placed in the blocked state and is entered on ESBQUEUE. The dispatcher is then called to select the next process to be run.

The SIGNAL request is made by a process that detects that some event corresponding to ESB has occurred. The algorithm for SIGNAL therefore records the event occurrence by setting ESBFLAG. It then scans ESBQUEUE, the list of processes waiting for this event. Each process on the list is moved from the blocked state to the ready state. Control is then returned to the process that made the SIGNAL request.

If the dispatching method being used is based on priorities, a slightly different SIGNAL algorithm is often used. On such systems, it may happen that one or more of the processes that were made ready has a higher priority than the currently running process. To take this into account, the SIGNAL algorithm would invoke the dispatcher instead of returning control directly to the requesting process. The dispatcher would then transfer control to the highest-priority process that is currently ready. This scheme is known as *preemptive* process scheduling. It permits a process that becomes ready to seize control from a lower-priority process that is currently running, without waiting for the time-slice of the lower-priority process to expire.

6.2.3 I/O Supervision

On a typical small computer, such as a standard SIC machine, input and output are usually performed 1 byte at a time. For example, a program that needs to read data might enter a loop that tests the status of the I/O device and executes a series of *read-data* instructions. On such systems, the CPU is involved with each byte of data being transferred to or from the I/O device. An example of this type of I/O programming can be found in Fig. 2.1.

More advanced computers often have special hardware to take care of the details of transferring data and controlling I/O devices. On SIC/XE, this function is performed by simple processors known as *I/O channels*. Figure 6.10 shows a typical I/O configuration for SIC/XE. There may be as many as 16 channels, and up to 16 devices may be connected to each channel. The

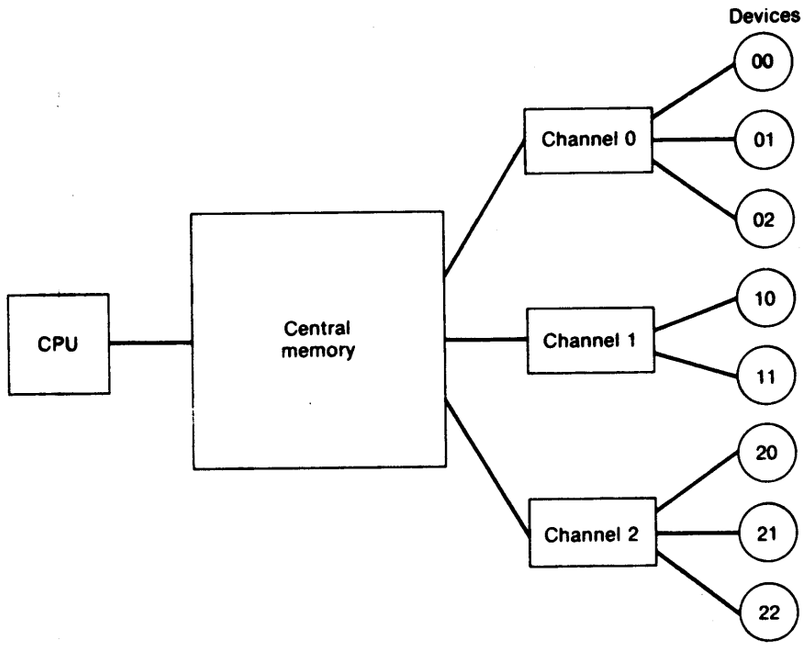


Figure 6.10 Typical I/O configuration for SIC/XE.

identifying number assigned to an I/O device also reflects the channel to which it is connected. For example, the devices numbered 20–2F are connected to channel 2.

The sequence of operations to be performed by a channel is specified by a *channel program,* which consists of a series of *channel commands.* To perform an I/O operation, the CPU executes a Start I/O (SIO) instruction, specifying a channel number and the beginning address of a channel program. The channel then performs the indicated I/O operation without further assistance from the CPU. After completing its program, the channel generates an I/O interrupt. Several channels can operate simultaneously, each executing its own channel program, so several different I/O operations can be in progress at the same time. Each channel operates independently of the CPU, so the CPU is free to continue computing while the I/O operations are carried out.

The operating system for a computer like SIC/XE is involved with the I/O process in several different ways. The system must accept I/O requests from user programs and inform these programs when the requested operations have been completed. It must also control the operation of the I/O channels and handle the I/O interrupts generated by the channels. In the remainder of

this section we discuss how these functions are performed and illustrate the process with several examples.

A SIC/XE program requests an I/O operation by executing an SVC 2 instruction. Parameters specify the channel number, the beginning address of a channel program, and the address of an event status block (ESB) that is used to signal completion of the I/O operation. When the program must wait for the results of an I/O operation, it executes an SVC 0 (WAIT) instruction. This instruction specifies the address of the ESB that corresponds to the I/O operation being awaited. Thus the general pattern for performing an I/O operation is

```
SVC 2    {request I/O operation}
  .
  .
  .
SVC 0    {wait for result}
```

In some cases, the WAIT may come immediately after the I/O request. However, because computing and I/O can be performed at the same time, it may be possible for the program to continue processing while awaiting the results of the I/O operation.

This procedure is illustrated in more detail by the program in Fig. 6.11. This program first loads the beginning address of a channel program, a channel number, and the address of an ESB into registers. The program then executes an SVC instruction to request the I/O operation. The channel program, defined as a sequence of data items, contains two channel commands. The first command specifies that a read operation is to be executed on device number 1 connected to the channel; 256 bytes of data are to be transferred into memory beginning at address BUFIN. The second command causes the channel to halt, which generates an I/O interrupt. The ESB consists of a 3-byte data area. The first bit of this ESB is a flag that is used to indicate whether or not the associated event has already occurred (0 = no, 1 = yes). The rest of the ESB is used to store a pointer to the queue of processes that are waiting for this event. If no processes are currently waiting, the pointer value is zero. Thus an initial ESB value of X'000000' indicates that the associated event has not yet occurred and that no processes are currently waiting for it. Further details concerning the format of SIC/XE channel commands can be found in Appendix C.

After issuing the I/O request, the program in Fig. 6.11 executes an SVC 0 instruction. Register A contains the address of the ESB that corresponds to the event being awaited, which in this case is the I/O operation just requested. After the read operation has been completed, the program moves the input

```
P1      START   0
        .
        .                       {initialization}
        .
        LDA     #READ           ADDRESS OF CHANNEL PROGRAM
        LDS     #1              CHANNEL NUMBER
        LDT     #ESB            ADDRESS OF EVENT STATUS BLOCK
        SVC     2               ISSUE READ REQUEST
LOOP    LDA     #ESB            ADDRESS OF ESB
        SVC     0               WAIT FOR COMPLETION OF READ
        .
        .                       {move data to program's work area}
        .
        LDA     #0              INITIALIZE ESB
        STA     ESB
        LDA     #READ
        LDS     #1
        LDT     #ESB
        SVC     2               ISSUE NEXT READ REQUEST
        .
        .                       {process data}
        .
        J       LOOP
.
.                               CHANNEL PROGRAM FOR READ
.                                  FIRST COMMAND--
READ    BYTE    X'11'                 COMMAND CODE = READ, DEVICE = 1
        BYTE    X'0100'               BYTE COUNT = 256
        WORD    BUFIN                 ADDRESS OF INPUT BUFFER
.                                  SECOND COMMAND--
        BYTE    X'000000000000'       HALT CHANNEL
.
ESB     BYTE    X'000000'       EVENT STATUS BLOCK FOR READ
BUFIN   RESB    256             BUFFER AREA FOR READ
        .
        .
        .
        END
```

Figure 6.11 Example of performing I/O using SVC requests.

data to a work area. It then re-initializes the ESB and requests an I/O operation to read the next 256 bytes of data. While this I/O operation is being performed, the program can process the data that has previously been read, thus overlapping the computation and input functions. After completing the processing of the previous data, the program returns to the top of its main loop to await the completion of the next read operation.

A slightly more complicated example is shown in Fig. 6.12. This program copies 4096-byte data records from device 22 to device 14. There are two channel programs, one for the read operation and one for the write, and two ESBs. The main loop of this program first issues a read request, and then waits for the completion of this read and for the completion of the previous write. After both operations are completed, the program builds the output record and issues the write request. It then returns to the top of the loop to read the next input record. On the first iteration of the loop, there has been no previous write request. At this time, however, the ESB for the write operation has its initially defined value of X'800000'. The first bit of this ESB has the value 1, indicating that the corresponding event has already occurred, so control is returned directly to the user program when the operating system WAIT routine is called (see Fig. 6.9).

Because the input and output operations for the program in Fig. 6.12 use different channels, these two operations are performed independently of each other. Either operation might be completed before the other. It is also possible that the two operations might actually be performed at the same time. The program is able to coordinate the related I/O operations because there is a different ESB corresponding to each operation. This program illustrates how I/O channels can be used to perform several overlapped I/O operations. Later in this section we consider a detailed example of this kind of overlap.

The programs in Figs. 6.11 and 6.12 illustrate I/O requests from the user's point of view. Now we are ready to discuss how such requests are actually handled by the operating system and the machine. The SIC/XE hardware provides a *channel work area* in memory corresponding to each I/O channel. This work area contains the starting address of the channel program currently being executed, if any, and the address of the ESB corresponding to the current operation. When an I/O operation is completed, the outcome is indicated by status flags that are stored in the channel work area. These flags indicate conditions such as normal completion, I/O error, or device unavailable. The channel work area also contains a pointer to a queue of I/O requests for the channel. This queue is maintained by the operating system routines. Appendix C contains additional details on the location and contents of the channel work areas for SIC/XE.

Figure 6.13 outlines the actions taken by the operating system in response to an I/O request from a user program. If the channel on which I/O is being requested is busy performing another operation, the operating system inserts the request into the queue for that channel. If the channel is not busy, the operating system stores the current request in the channel work area and starts the channel. In either case, control is then returned to the process that made the I/O request so that it can continue to execute while the I/O is being performed.

```
P2      START   0
        .
        .                       {initialization}
        .
LOOP    LDA     #0              INITIALIZE ESB FOR READ
        STA     RDESB
        LDA     #READ           ISSUE READ REQUEST FOR DEVICE 22
        LDS     #2
        LDT     #RDESB
        SVC     2
        LDA     #RDESB          WAIT FOR COMPLETION OF READ
        SVC     0
        LDA     #WRESB          WAIT FOR COMPLETION OF PREVIOUS WRITE
        SVC     0
        .
        .                       {build output record}
        .
        LDA     #0              INITIALIZE ESB FOR WRITE
        STA     WRESB
        LDA     #WRITE          ISSUE WRITE REQUEST FOR DEVICE 14
        LDS     #1
        LDT     #WRESB
        SVC     2
        J       LOOP
.
.                               CHANNEL PROGRAM FOR READ
.                                   FIRST COMMAND--
READ    BYTE    X'12'                   COMMAND = READ, DEVICE = 2
        BYTE    X'1000'                 BYTE COUNT = 4096
        WORD    BUFIN                   ADDRESS OF INPUT BUFFER
.                                   SECOND COMMAND--
        BYTE    X'000000000000'         HALT CHANNEL
.
.                               CHANNEL PROGRAM FOR WRITE
.                                   FIRST COMMAND--
WRITE   BYTE    X'24'                   COMMAND = WRITE, DEVICE = 4
        BYTE    X'1000'                 BYTE COUNT = 4096
        WORD    BUFOUT                  ADDRESS OF OUTPUT BUFFER
.                                   SECOND COMMAND--
.       BYTE    X'000000000000'          HALT CHANNEL
RDESB   BYTE    X'000000'       EVENT STATUS BLOCK FOR READ
WRESB   BYTE    X'800000'       EVENT STATUS BLOCK FOR WRITE
BUFIN   RESB    4096            INPUT BUFFER
BUFOUT  RESB    4096            OUTPUT BUFFER
        .
        .
        .
        END
```

Figure 6.12 Program illustrating multiple I/O requests.

```
procedure IOREQ (CHAN, CP, ESB)

    test channel using TIO
    if channel is busy then
        insert (CP,ESB) on queue for channel
    else
        begin
            store (CP,ESB) in channel work area
            start channel using SIO
        end
    return control to requesting process using LPS
```

Figure 6.13 Algorithm for processing an I/O request (SVC 2).

Figure 6.14 describes the actions taken by the operating system in response to an I/O interrupt. The number of the I/O channel that generated the interrupt can be found in the status word that is stored in the I/O-interrupt work area. The interrupt-handling routine then examines the status flags in the work area for this channel to determine the cause of the interrupt.

If the channel status flags indicate normal completion of the I/O operation, the interrupt handler signals this completion via the ESB that was specified in the I/O request. This may be done either by making an SVC request, which results in a nested interrupt situation, or by directly invoking the part of the operating system that processes SIGNAL requests. In either case, the ESB is marked to indicate completion of the I/O operation. Any process that had previously been awaiting this completion is returned to the ready state (see Section 6.2.2). The I/O-interrupt handler then examines the queue of pending requests for this channel and starts the channel performing the next request, if any.

If the channel status flags indicate some abnormal condition, the operating system initiates the appropriate error-recovery action. This action, of course, depends upon the nature of the I/O device and the error detected. For example, a parity error on a magnetic tape device is normally handled by backspacing the tape and restarting the I/O operation (up to some maximum number of times). On the other hand, an indication that a line printer is out of paper is handled by issuing a message to the computer operator before attempting any further recovery. If the operating system determines that an I/O error is uncorrectable, it may terminate the process that made the I/O request and send an appropriate message to the user. Alternatively, it might signal completion of the operation and store an error code in the ESB. The requesting process could then make its own decision about whether or not to continue.

After its processing is complete, the interrupt handler ordinarily returns control by restoring the status of the interrupted process. However, if the CPU

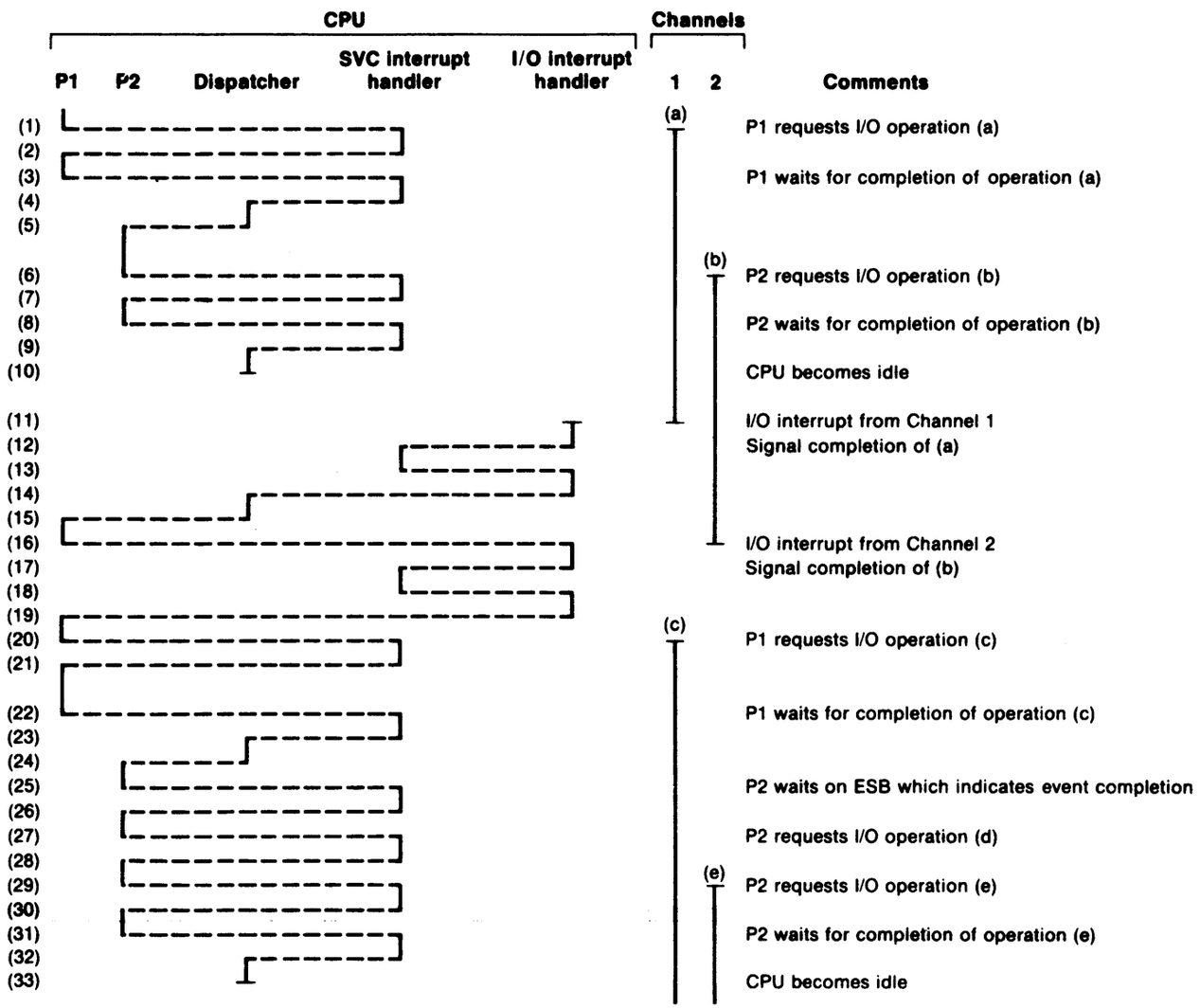
```
procedure IOINTERRUPT(CHAN)

    examine status flags in channel work area
    if normal completion of operation then
        begin
            get ESB address from channel work area
            use SVC to signal occurrence of event for ESB
            if request queue for channel is not empty then
                begin
                    get (CP,ESB) for next request from queue
                    store (CP, ESB) in channel work area
                    start channel using SIO
                end {if not empty}
        end {if normal completion}
    else
        take appropriate error recovery action
    if CPU was in idle state when the interrupt occurred then
        DISPATCH
    else
        return control to interrupted process using LPS
```

Figure 6.14 Algorithm for processing an I/O interrupt.

was idle at the time of the interrupt, the dispatcher must be invoked. This is because the interrupt processing may have caused a process to become ready to execute. If this is the case, the CPU should not be restored to an idle status. Likewise, the dispatcher would be invoked if preemptive process scheduling is being used (see Section 6.2.2).

Figure 6.15 provides an illustration of the process-scheduling and I/O-supervision functions we have described. Two user processes, designated P1 and P2, are being executed concurrently. These are the same processes that are outlined in Figs. 6.11 and 6.12. We assume the time-slice provided to each process by the dispatcher is relatively large so that timer interrupts do not ordinarily occur before the process must give up control for some other reason. The diagram in Fig. 6.15 shows the flow of activities being performed by the CPU, divided between the user processes and the parts of the operating system, and by the two I/O channels. The time scale runs from top to bottom on the diagram. Distances on this scale are not necessarily proportional to the actual lengths of time involved. Sequence numbers are provided to aid in the use of this example.

At the beginning of the example, processes P1 and P2 have both been initiated, and P1 has been dispatched first. Both I/O channels are idle. At sequence number (1), P1 makes its first I/O request by executing an SVC instruction. This causes an interrupt, which transfers control to the SVC-interrupt handler. For ease of reference, this I/O operation is designated in the diagram by (a).


CPU
Channels
P1
P2
Dispatcher
SVC interrupt handler
I/O interrupt handler
1
2
Comments
(1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)
(9)
(10)
(11)
(12)
(13)
(14)
(15)
(16)
(17)
(18)
(19)
(20)
(21)
(22)
(23)
(24)
(25)
(26)
(27)
(28)
(29)
(30)
(31)
(32)
(33)
(a)
(b)
(c)
(e)
P1 requests I/O operation (a)
P1 waits for completion of operation (a)
P2 requests I/O operation (b)
P2 waits for completion of operation (b)
CPU becomes idle
I/O interrupt from Channel 1
Signal completion of (a)
I/O interrupt from Channel 2
Signal completion of (b)
P1 requests I/O operation (c)
P1 waits for completion of operation (c)
P2 waits on ESB which indicates event completion
P2 requests I/O operation (d)
P2 requests I/O operation (e)
P2 waits for completion of operation (e)
CPU becomes idle

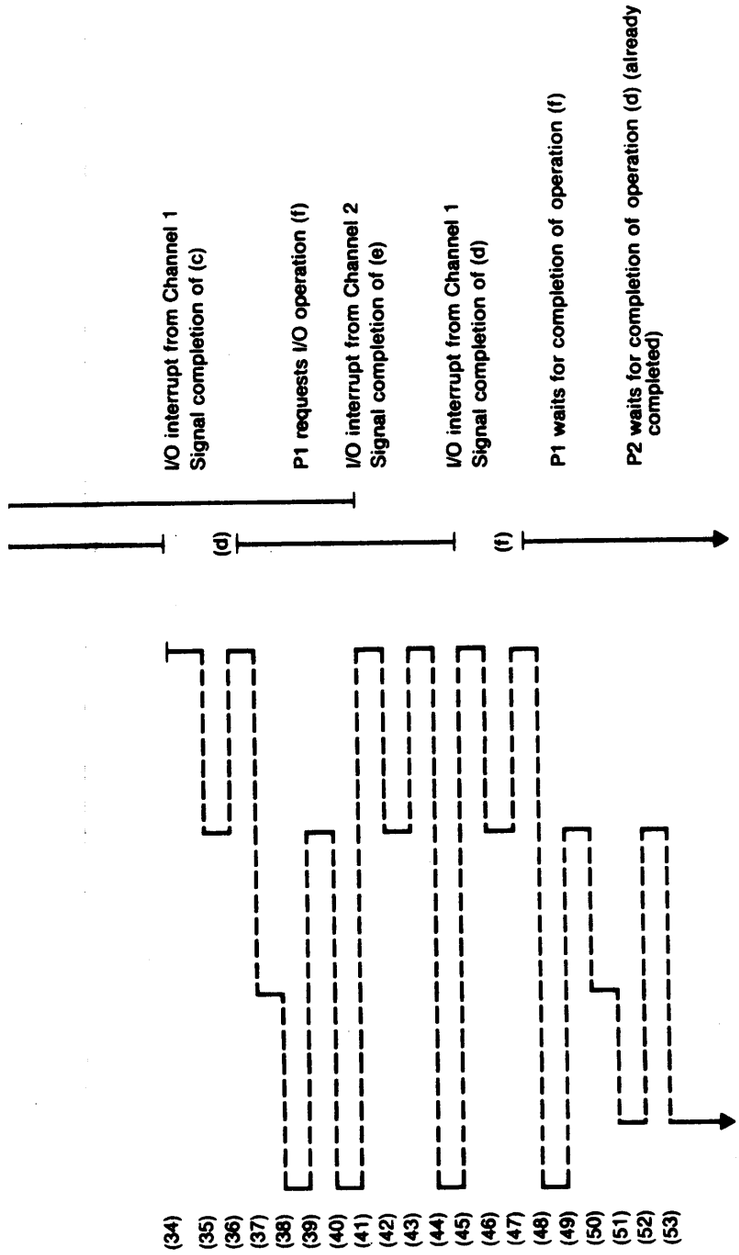


Figure 6.15 Example of I/O-supervision and process-scheduling functions.

The I/O request specifies channel 1, which is currently free. Therefore, the SVC-interrupt handler starts the channel program and returns control to process P1 (sequence (2)).

At (3), P1 issues a WAIT request for operation (a) by executing another SVC instruction (SVC 0). Control is once again transferred to the SVC-interrupt handler. The ESB specified in this WAIT request indicates that the associated event has not yet occurred. Therefore, process P1 is placed in the blocked state, and the dispatcher is invoked (4). The dispatcher then switches control to process P2 (5). At sequence number (6), P2 issues its first I/O request, which is for device 2 on channel 2. Since channel 2 is free, the I/O operation is started, and control returns to P2. At (8), P2 must wait for the completion of its I/O request; since this event has not yet occurred, P2 becomes blocked. At (9), the dispatcher is invoked as before. This time, however, both processes are blocked, so the dispatcher places the CPU into an idle state (10). Note that both I/O channels are still active.

The CPU remains idle until (11), when channel 1 completes its I/O operation. We assume in this example that all operations are completed normally. The channel generates an I/O interrupt, which switches the CPU from its idle state to the I/O-interrupt handler. After determining that the operation was completed normally, the I/O-interrupt handler issues a SIGNAL request (SVC 1) for the associated ESB. This switches control to the SVC-interrupt handler (12). Process P1, which is waiting on this ESB, is placed in the ready state. The SVC handler then returns control to the I/O-interrupt handler (13). The dispatcher is invoked at sequence (14), and switches control to process P1 at (15). A similar series of operations occurs when channel 2 completes its operation (16); this causes process P2 to be made ready. However, since the CPU was not idle at the time of the interrupt, control does not pass immediately to P2. The I/O-interrupt handler restores control to the interrupted process P1. P2 does not receive control until P1 issues its next WAIT request at sequence (22).

You should carefully follow through the other steps in this example to be sure you understand the flow of control in response to the various interrupts. In doing this, you may find it useful to refer to the algorithms in Figs. 6.8, 6.9, 6.13, and 6.14. Note in particular the many different types of overlap between the CPU execution and the I/O operations of the different processes. The ability to provide such flexible sequencing of tasks is one of the most important advantages of an interrupt-driven operating system.

6.2.4 Management of Real Memory

Any operating system that supports more than one user at a time must provide a mechanism for dividing central memory among the concurrent processes.

Many multiprogramming and multiprocessing systems divide memory into *partitions*, with each process being assigned to a different partition. These partitions may be predefined in size and position (*fixed partitions*), or they may be allocated dynamically according to the requirements of the jobs being executed (*variable partitions*).

When variable partitions are used, it is not necessary to select partition sizes in advance. However, the operating system must do more work in keeping track of which areas of memory are allocated and which areas are free. Usually the system does this by maintaining a linked list of free memory areas. This list is scanned when a new partition is to be allocated. The partition is placed either in the first free area in which it will fit (*first-fit* allocation), or in the smallest free area in which it will fit (*best-fit* allocation). When a partition is released, its assigned memory is returned to the free list and combined with any adjacent free areas. A detailed discussion and comparison of such dynamic storage allocation algorithms can be found in Lewis and Denenberg (1991).

Regardless of the partitioning technique that is used, it is necessary for the operating system and the hardware to provide *memory protection*. When a job is running in one partition, it must be prevented from modifying memory locations in any other partition or in the operating system. Some systems allow the reading of data anywhere in memory, but permit writing only within a job's own partition. Other systems restrict both reading and writing to the job's partition.

Some type of hardware support is necessary for effective memory protection. One simple scheme provides a pair of *bounds registers* that contain the beginning and ending addresses of a job's partition. These registers are not directly accessible to user programs; they can be accessed only when the CPU is in supervisor mode. The operating system sets the bounds registers when a partition is assigned to a user job. The values in these registers are automatically saved and restored during context switching operations such as those caused by an interrupt or an LPS instruction. Thus the bounds registers always contain the beginning and ending addresses of the partition assigned to the currently executing process. For every memory reference, the hardware automatically checks the referenced address against the bounds registers. If the address is outside the current job's partition, the memory reference is not performed and a program interrupt is generated.

A different type of memory protection scheme is used on SIC/XE. Each 800-byte (hexadecimal) block of memory has associated with it a 4-bit *storage protection key*. These keys can be set by the operating system using the privileged instruction SSK (Set Storage Key). Each user process has assigned to it a 4-bit *process identifier*, which is stored in the ID field of the status word SW. When a partition is assigned to a job, the operating system sets the storage keys for all blocks of memory within the partition to the value of the process identifier for that job. For each memory reference by a user program, the hardware

automatically compares the process identifier from SW to the protection key for the block of memory being addressed. If the values of these two fields are not the same, the memory reference is not performed and a program interrupt is generated. However, this test is not performed when the CPU is in supervisor mode; the operating system is allowed to reference any location in memory.

One problem common to all general-purpose dynamic storage allocation techniques is *memory fragmentation.* Fragmentation occurs when the available free memory is split into several separate blocks, with each block being too small to be of use.

Figure 6.16 illustrates one possible solution to this problem: *relocatable partitions.* After each job terminates, the remaining partitions are moved as far as possible toward one end of memory. This movement gathers all the available free memory together into one contiguous block that is more useful for allocating new partitions.

In practice, the implementation of relocatable partitions requires some hardware support. There is a special *relocation register* that is set by the operating system to contain the beginning address of the program currently being executed. This register is automatically saved and restored during context switching operations, and its value is modified by the operating system when a program is

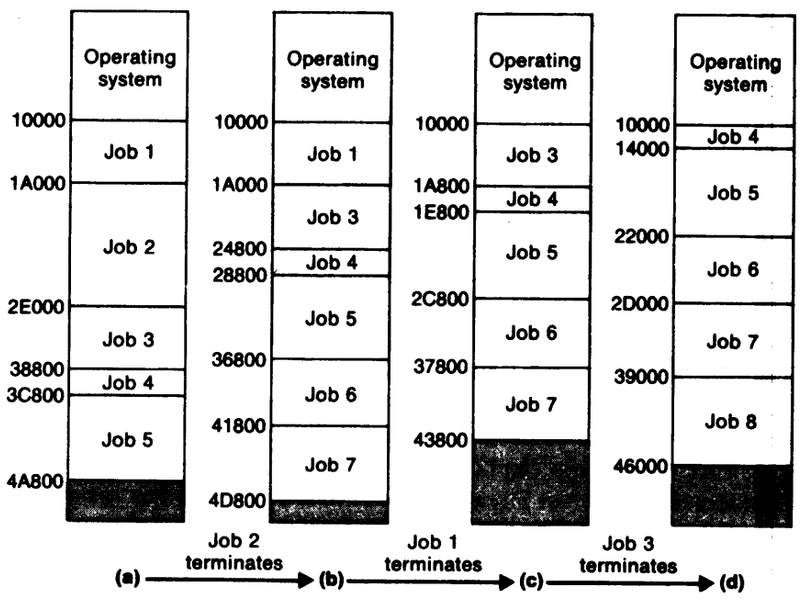


Figure 6.16 Memory allocation for jobs using relocatable partitions.

moved to a new location. The value in the relocation register is automatically added to the address for every memory reference made by the user program.

It is important to understand that the relocation register is under the control of the operating system; it is not directly available to the user program. Thus this register is quite different from the programmer-defined base registers found on SIC/XE, PowerPC, and many other computers. The relocation register is *automatically* involved each time a program refers to any location in memory, so it provides exactly the same effect as if each program were really loaded at actual address 00000. Indeed, on many computers there is no direct way for a user program to determine where it is actually located in memory. Thus this type of relocation applies to addresses given by pointers in data structures and values in base registers as well as to addresses in instructions. Note also that this automatic relocation performed by the hardware eliminates the need for relocation when the program is loaded.

In this section we described methods for allocating program partitions of predetermined size. Some operating systems allow user programs to dynamically request additional memory during execution. The additional memory assigned need not necessarily be contiguous with the original partition. Such dynamic storage allocation is usually performed with methods similar to those used in managing storage for data structures. A good discussion of such techniques can be found in Lewis and Denenberg (1991).

6.2.5 Management of Virtual Memory

A *virtual* resource is one that appears to a user program to have characteristics that are different from those of the actual implementation of the resource. User programs are allowed to use a large contiguous *virtual memory*, sometimes called a *virtual address space*. This virtual memory may even be larger than the total amount of real memory available on the computer. The virtual memory used by a program is stored on some external device (the *backing store*). Portions of this virtual memory are mapped into real memory as they are needed by the program. The backing store and the virtual-to-real mapping are completely invisible to the user program. The program is written exactly as though the virtual memory really existed.

In this section we describe *demand paging*, which is one common method for implementing virtual memory. References to discussions of other types of virtual memories can be found at the end of the section.

In a typical demand-paging system, the virtual memory of a process is divided into *pages* of some fixed length. The real memory of the computer is divided into *page frames* of the same length as the pages. Any page from any process can potentially be loaded into any page frame in real memory.

The mapping of pages onto page frames is described by a *page map table* (PMT); there is one PMT for each process in the system. The PMT is used by the hardware to convert addresses in a program's virtual memory into the corresponding addresses in real memory. This conversion process is similar to the use of the relocation register described in the last section. However, there is one PMT entry for each page instead of one relocation register for the entire program. This conversion of virtual addresses to real addresses is known as *dynamic address translation.*

These concepts are illustrated by the program outlined in Fig. 6.17. The program is divided into pages that are 1000 bytes (hexadecimal) in length. Virtual addresses 0000 through 0FFF are in Page 0; addresses 1000 through 1FFF are in Page 1, and so on. When the execution of the program is begun, the operating system loads Page 0, the page containing the first executable instruction, into some page frame in real memory. Other pages are loaded into memory as they are needed.

The processes of dynamic address translation and page loading are illustrated in Fig. 6.18. In Fig. 6.18(a), Page 0 of the program has been loaded into page frame 1D (i.e., real-memory addresses 1D000–1DFFF). Consider first the JEQ instruction that is located at virtual address 0103. The operand address for this instruction is virtual address 0420. We used an instruction format that provides direct addressing to make this initial example easier to follow. The operand address 0420 is located within Page 0, at offset 420 from the beginning of the page. The page map table indicates that Page 0 of this program is loaded in page frame 1D (that is, beginning at address 1D000). Thus the real address calculated by the dynamic address translation is 1D420.

Next let us consider the LDA instruction at virtual address 0420. The operand for this instruction is at virtual address 6FFA (Page 6, offset FFA). However, Page 6 has not yet been loaded into real memory, so the dynamic address translation hardware is not able to compute a real address. Instead, it generates a special type of program interrupt called a *page fault* [see Fig. 6.18(b)]. The interrupt-handling routine, which we discuss later, responds to this interrupt by loading the required page into some page frame. Let us assume page frame 29 is chosen. The instruction that caused the interrupt is then reexecuted. This time, as shown in Fig. 6.18(c), the dynamic address translation is successful.

The other pages of the program are loaded on demand in a similar way. Assume that Fig. 6.17 shows all the Jump instructions in the program as well as all instructions whose operands are located in another page. When control passes from the last instruction in Page 0 to the first instruction in Page 1, the instruction-fetch operation causes a page fault, which results in the loading of Page 1. The STA instruction at virtual address 1840 causes Page 8 to be loaded. Page 2 is then loaded as a result of the instruction-fetch cycle, just as Page 1

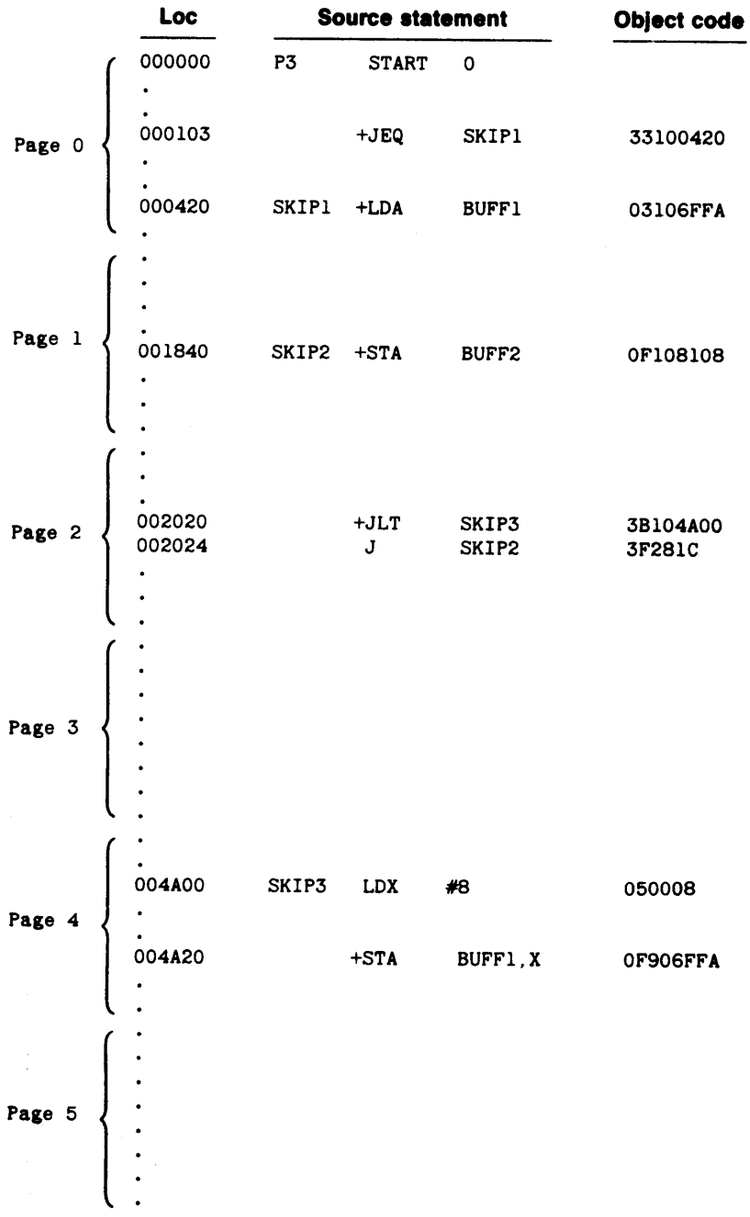


Figure 6.17 Program for illustration of demand paging.

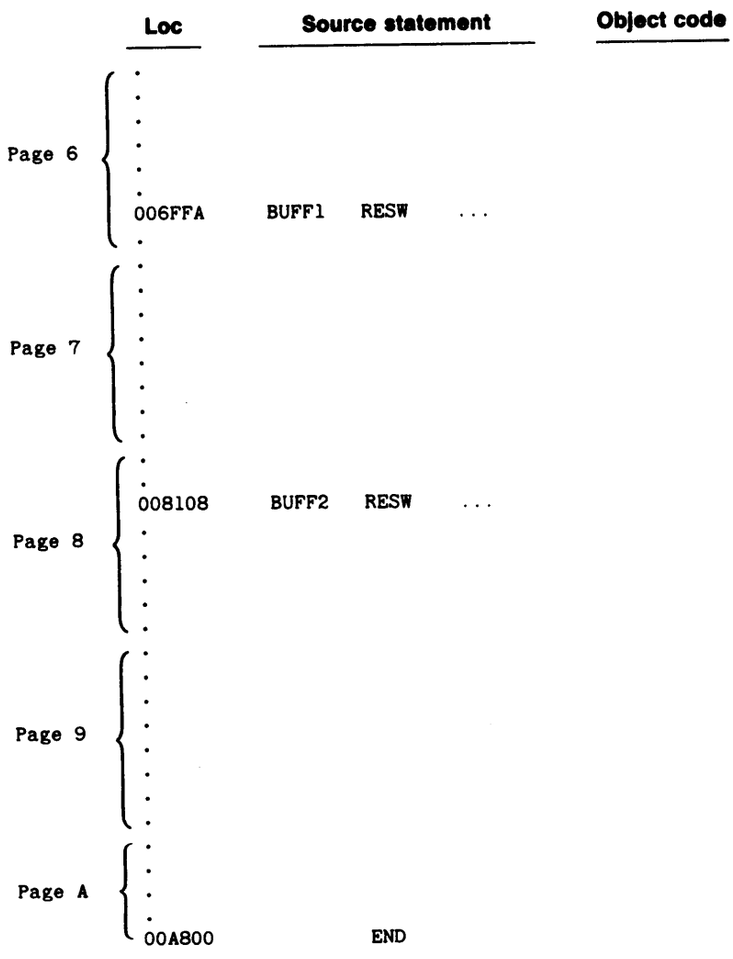


Figure 6.17 (*cont'd*)

was. Now consider the two Jump instructions at addresses 2020 and 2024. If the first of these jumps is executed (i.e., if the less-than condition is true), it causes Page 4 to be loaded; otherwise, Page 4 remains unloaded. In this latter case, the unconditional jump at 2024 transfers control back to a location in Page 1 (which has already been loaded). After control passes to Page 4, the STA instruction at 4A20 is executed. This instruction specifies an operand address of 6FFA, with indexed addressing. We assume the value 8 remains in the index register. The resulting target address is 7002; as a result of this instruction, Page 7 is loaded.

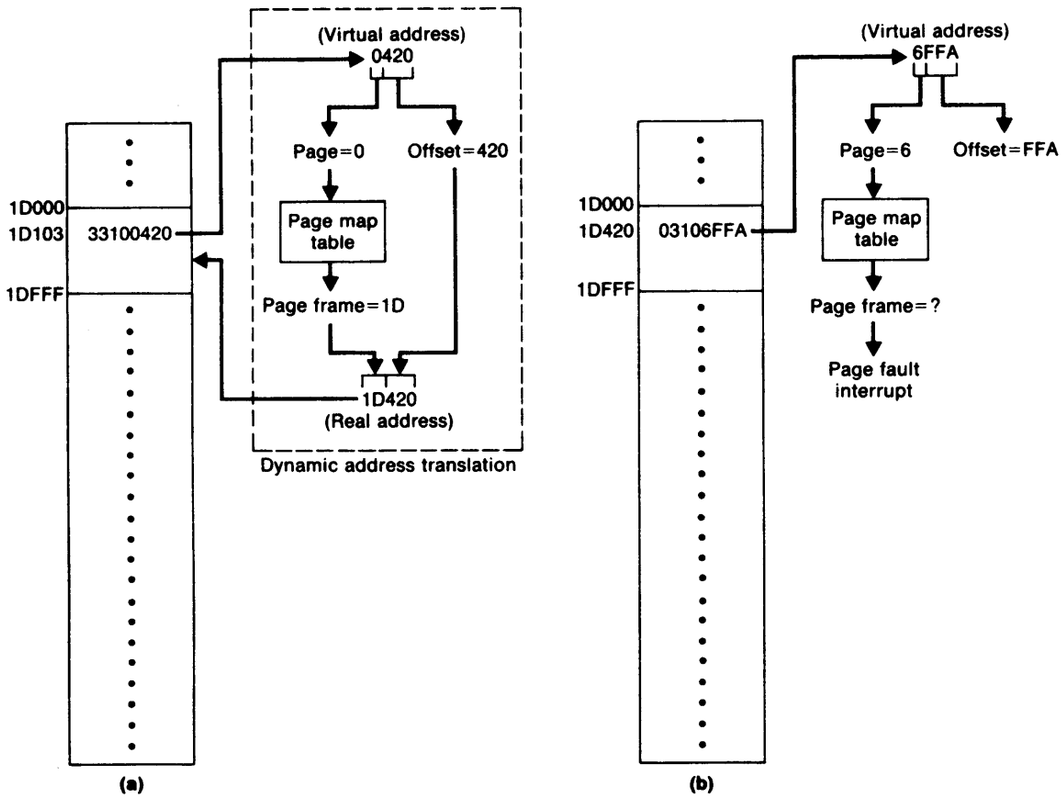


Figure 6.18 Examples of dynamic address translation and demand paging.

Figure 6.19 shows the situation after the sequence of events just described. The page map table for P3 reflects the fact that Pages 0, 1, 2, 4, 6, 7, and 8 are currently loaded, and gives the corresponding page frame numbers. There is a similar page table for every other program in the system. Note that the PMT also indicates which pages have been modified since they were loaded (in this case, Pages 7 and 8). This information is used by the page-fault interrupt-handling routine when it is necessary to remove a page already in memory.

Figure 6.20 summarizes the address-translation and demand-paging functions illustrated in the previous discussion. Figure 6.20(a) describes the dynamic address translation algorithm used. Recall that this algorithm is implemented directly by the hardware of the machine. If the dynamic address translation cannot be completed because the required page is not in memory, a page fault interrupt occurs.

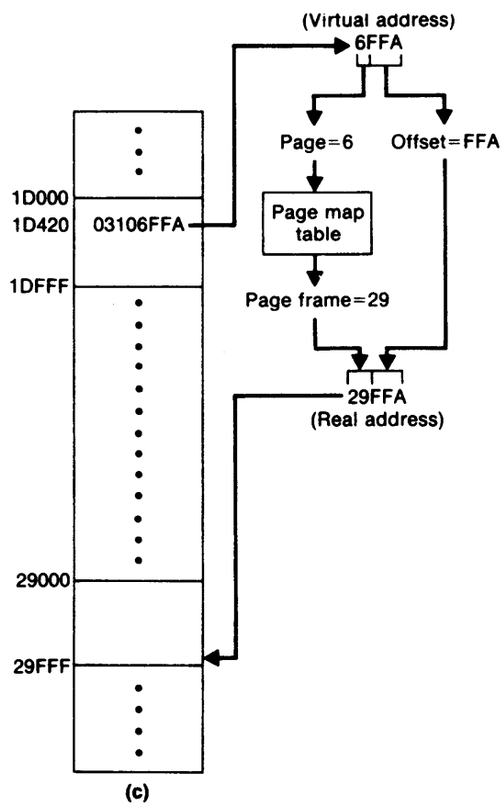


Figure 6.18 (*cont'd*)

Figure 6.20(b) describes the interrupt-handling routine that is invoked for a page fault. The operating system maintains a table describing the status of all page frames. The first step in processing a page fault interrupt is to search this table for an empty page frame. If an empty page frame is found, the required page can be loaded immediately. Otherwise, a page currently in memory must be removed to make room for the page to be loaded. If the page being removed has been modified since it was last loaded, then the updated version must be rewritten on the backing store. If the page has not been modified, the image in memory can simply be discarded.

The page fault interrupt-handling routine described in Fig. 6.20(b) requires the performance of at least one physical I/O operation. Thus this routine takes much longer to execute than any of the other interrupt handlers we have discussed. It is usually not desirable to inhibit interrupts for long periods of