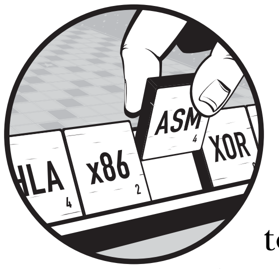
7

LOW-LEVEL CONTROL STRUCTURES



This chapter discusses "pure" assembly language control statements. You'll need to master these low-level control structures before you can claim to be an assembly language programmer. By the time you finish this chapter, you should be able to stop using HLA's high-level control statements and synthesize them using low-level 80x86 machine instructions.

The last section of this chapter discusses *hybrid* control structures that combine the features of HLA's high-level control statements with the 80x86 control instructions. These combine the power and efficiency of the low-level control statements with the readability of high-level control statements. Advanced assembly programmers may want to use these hybrid statements to improve their programs' readability without sacrificing efficiency.

7.1 Low-Level Control Structures

Until now, most of the control structures you've seen and have used in your programs are similar to the control structures found in high-level languages like Pascal, C++, and Ada. While these control structures make learning assembly language easy, they are not true assembly language statements. Instead, the HLA compiler translates these control structures into a sequence of "pure" machine instructions that achieve the same result as the high-level control structures. This text uses the high-level control structures to allow you to learn assembly language without having to learn everything all at once. Now, however, it's time to put aside these high-level control structures and learn how to write your programs in *real* assembly language, using low-level control structures.

7.2 Statement Labels

Assembly language low-level control structures make extensive use of *labels* within your source code. A low-level control structure usually transfers control between two points in your program. You typically specify the destination of such a transfer using a statement label. A statement label consists of a valid (unique) HLA identifier and a colon. For example:

```
aLabel:
```

Of course, as for procedure, variable, and constant identifiers, you should attempt to choose descriptive and meaningful names for your labels. The example identifier above, `aLabel`, is hardly descriptive or meaningful.

Statement labels have one important attribute that differentiates them from most other identifiers in HLA: You don't have to declare a label before you use it. This is important, because low-level control structures must often transfer control to some point later in the code; therefore the label may not be defined by the time you reference it.

You can do three things with labels: transfer control to a label via a jump (`goto`) instruction, call a label via the `call` instruction, and take the address of a label. There is very little else you can directly do with a label (of course, there is very little else you would want to do with a label, so this is hardly a restriction). The program in Listing 7-1 demonstrates two ways to take the address of a label in your program and print out the address (using the `lea` instruction and using the `&` address-of operator):

```
program labelDemo;
#include( "stdlib.hhf" );

begin labelDemo;

    lbl1:

        lea( ebx, lbl1 );
        mov( &lbl2, eax );
        stdout.put( "&lbl1=$", ebx, " &lbl2=", eax, nl );
    lbl2:

end labelDemo;
```

Listing 7-1: Displaying the address of statement labels in a program

HLA also allows you to initialize double-word variables with the addresses of statement labels. However, there are some restrictions on labels that appear in the initialization portions of variable declarations. The most important restriction is that you must define the statement label at the same lexical level as the variable declaration. That is, if you reference a statement label in the initializer of a variable declaration appearing in the main program, the statement label must also be in the main program. Conversely, if you take the address of a statement label in a local variable declaration, that symbol must appear in the same procedure as the local variable. Listing 7-2 demonstrates the use of statement labels in variable initialization:

```
program labelArrays;
#include( "stdlib.hhf" );

static
    labels:dword[2] := [ &lbl1, &lbl2 ];

    procedure hasLabels;
    static
        stmtLbls: dword[2] := [ &label1, &label2 ];

    begin hasLabels;

        label1:

            stdout.put
            (
                "stmtLbls[0]= $", stmtLbls[0], nl,
                "stmtLbls[1]= $", stmtLbls[4], nl
            );

        label2:

    end hasLabels;

begin labelArrays;

    hasLabels();
    lbl1:

        stdout.put( "labels[0]= $", labels[0], " labels[1]=", labels[4], nl );

    lbl2:

end labelArrays;
```

Listing 7-2: Initializing dword variables with the address of statement labels

Once in a while, you'll need to refer to a label that is not within the current procedure. The need for this is sufficiently rare that this text will not describe all the details. See the HLA documentation for more details should you ever need to do this.

7.3 Unconditional Transfer of Control (jmp)

The `jmp` (jump) instruction unconditionally transfers control to another point in the program. There are three forms of this instruction: a direct jump and two indirect jumps. These instructions take the following forms:

```
    jmp label;
    jmp( reg32 );
    jmp( mem32 );
```

The first instruction is a direct jump above. For direct jumps you normally specify the target address using a statement label. The label appears either on the same line as an executable machine instruction or by itself on a line preceding an executable machine instruction. The direct jump is completely equivalent to a `goto` statement in a high-level language.[1]

Here's an example:

```
        << statements >>
        jmp laterInPgm;
            .
            .
            .
laterInPgm:
        << statements >>
```

The second form of the `jmp` instruction given earlier—`jmp( reg32 );`—is a *register indirect* jump instruction. This instruction transfers control to the instruction whose address appears in the specified 32-bit general-purpose register. To use this form of the `jmp` instruction, you must load a 32-bit register with the address of some machine instruction prior to the execution of the `jmp`. You could use this instruction to implement a *state machine* by loading a register with the address of some label at various points throughout your program and then use a single indirect jump at a common point to transfer control to one of those labels. The short sample program in Listing 7-3 demonstrates how you could use the `jmp` in this manner.

```
program regIndJmp;
#include( "stdlib.hhf" );

static
    i:int32;
```

[1] Unlike high-level languages, where your instructors usually forbid you to use `goto` statements, you will find that the use of the `jmp` instruction in assembly language is essential.

```
begin regIndJmp;

    // Read an integer from the user and set ebx to
    // denote the success or failure of the input.

    try

        stdout.put( "Enter an integer value between 1 and 10: " );
        stdin.get( i );
        mov( i, eax );
        if( eax in 1..10 ) then

            mov( &GoodInput, ebx );

        else

            mov( &valRange, ebx );

        endif;

      exception( ex.ConversionError )

        mov( &convError, ebx );

      exception( ex.ValueOutOfRange )

        mov( &valRange, ebx );

    endtry;

    // Okay, transfer control to the appropriate
    // section of the program that deals with
    // the input.

    jmp( ebx );

    valRange:
        stdout.put( "You entered a value outside the range 1..10" nl );
        jmp Done;

    convError:
        stdout.put( "Your input contained illegal characters" nl );
        jmp Done;

    GoodInput:
        stdout.put( "You entered the value ", i, nl );

    Done:


end regIndJmp;
```

Listing 7-3: Using register-indirect `jmp` instructions

The third form of the `jmp` instruction given earlier is a memory-indirect `jmp`. This form of the `jmp` instruction fetches the double-word value from the memory location and jumps to that address. This is similar to the register-indirect `jmp` except the address appears in a memory location rather than in a register. Listing 7-4 demonstrates a rather trivial use of this form of the `jmp` instruction.

```
program memIndJmp;
#include( "stdlib.hhf" );

static
    LabelPtr:dword := &stmtLabel;

begin memIndJmp;

    stdout.put( "Before the JMP instruction" nl );
    jmp( LabelPtr );

        stdout.put( "This should not execute" nl );

    stmtLabel:

        stdout.put( "After the LabelPtr label in the program" nl );

end memIndJmp;
```

Listing 7-4: Using memory-indirect `jmp` instructions

WARNING *Unlike the HLA high-level control structures, the low-level `jmp` instructions can cause you a lot of trouble. In particular, if you do not initialize a register with the address of a valid instruction and you jump indirectly through that register, the results are undefined (though this will usually cause a general protection fault). Similarly, if you do not initialize a double-word variable with the address of a legal instruction, jumping indirectly through that memory location will probably crash your program.*

7.4 The Conditional Jump Instructions

Although the `jmp` instruction provides transfer of control, it is inconvenient to use when making decisions such as those you'll need to implement statements like `if` and `while`. The 80x86's conditional jump instructions handle this task.

The conditional jumps test one or more CPU flags to see if they match some particular pattern. If the flag settings match the condition, the conditional jump instruction transfers control to the target location. If the match fails, the CPU ignores the conditional jump and execution continues with the instruction following the conditional jump. Some conditional jump instructions simply test the setting of the sign, carry, overflow, and zero flags. For example, after the execution of a `shl` instruction, you could test the carry flag to determine if the `shl` shifted a 1 out of the H.O. bit of its operand. Likewise, you could test the zero flag after a `test` instruction to check if the result was 0. Most of the time, however, you will probably execute a conditional jump after a `cmp` instruction. The `cmp` instruction sets the flags so that you can test for less than, greater than, equality, and so on.

The conditional `jmp` instructions take the following form:

```
jcc label;
```

The *cc* in j*cc* indicates that you must substitute some character sequence that specifies the type of condition to test. These are the same characters the set*cc* instruction uses. For example, `js` stands for *jump* if the sign flag is set. A typical `js` instruction is:

```
js ValueIsNegative;
```

In this example, the `js` instruction transfers control to the `ValueIsNegative` label if the sign flag is currently set; control falls through to the next instruction following the `js` instruction if the sign flag is clear.

Unlike the unconditional `jmp` instruction, the conditional jump instructions do not provide an indirect form. They only allow a branch to a statement label in your program.

NOTE *Intel's documentation defines various synonyms or instruction aliases for many conditional jump instructions.*

Tables 7-1, 7-2, and 7-3 list all the aliases for a particular instruction. These tables also list the opposite branches. You'll soon see the purpose of the opposite branches.

Table 7-1: j*cc* Instructions That Test Flags

Instruction	Description	Condition	Aliases	Opposite
jc	Jump if carry	Carry = 1	jb, jnae	jnc
jnc	Jump if no carry	Carry = 0	jnb, jae	jc
jz	Jump if zero	Zero = 1	je	jnz
jnz	Jump if not zero	Zero = 0	jne	jz
js	Jump if sign	Sign = 1		jns
jns	Jump if no sign	Sign = 0		js
jo	Jump if overflow	Overflow = 1		jno
jno	Jump if no overflow	Overflow = 0		jo
jp	Jump if parity	Parity = 1	jpe	jnp
jpe	Jump if parity even	Parity = 1	jp	jpo
jnp	Jump if no parity	Parity = 0	jpo	jp
jpo	Jump if parity odd	Parity = 0	jnp	jpe

Table 7-2: `jcc` Instructions for Unsigned Comparisons

Instruction	Description	Condition	Aliases	Opposite
`ja`	Jump if above (>)	Carry = 0, Zero = 0	`jnbe`	`jna`
`jnbe`	Jump if not below or equal (`not <=`)	Carry = 0, Zero = 0	`ja`	`jbe`
`jae`	Jump if above or equal (>=)	Carry = 0	`jnc, jnb`	`jnae`
`jnb`	Jump if not below (`not <`)	Carry = 0	`jnc, jae`	`jb`
`jb`	Jump if below (<)	Carry = 1	`jc, jnae`	`jnb`
`jnae`	Jump if not above or equal (`not >=`)	Carry = 1	`jc, jb`	`jae`
`jbe`	Jump if below or equal (<=)	Carry = 1 or Zero = 1	`jna`	`jnbe`
`jna`	Jump if not above (`not >`)	Carry = 1 or Zero = 1	`jbe`	`ja`
`je`	Jump if equal (=)	Zero = 1	`jz`	`jne`
`jne`	Jump if not equal (¦)	Zero = 0	`jnz`	`je`

Table 7-3: `jcc` Instructions for Signed Comparisons

Instruction	Description	Condition	Aliases	Opposite
`jg`	Jump if greater (>)	Sign = Overflow or Zero = 0	`jnle`	`jng`
`jnle`	Jump if not less than or equal (`not <=`)	Sign = Overflow or Zero = 0	`jg`	`jle`
`jge`	Jump if greater than or equal (>=)	Sign = Overflow	`jnl`	`jge`
`jnl`	Jump if not less than (`not <`)	Sign = Overflow	`jge`	`jl`
`jl`	Jump if less than (<)	Sign <> Overflow	`jnge`	`jnl`
`jnge`	Jump if not greater or equal (`not >=`)	Sign <> Overflow	`jl`	`jge`
`jle`	Jump if less than or equal (<=)	Sign <> Overflow or Zero = 1	`jng`	`jnle`
`jng`	Jump if not greater than (`not >`)	Sign <> Overflow or Zero = 1	`jle`	`jg`
`je`	Jump if equal (=)	Zero = 1	`jz`	`jne`
`jne`	Jump if not equal (¦)	Zero = 0	`jnz`	`je`

One brief comment about the Opposite column is in order. In many instances you will need to be able to generate the opposite of a specific branch instruction (examples appear later in this section). With only two exceptions, a very simple rule completely describes how to generate an opposite branch:

- If the second letter of the `jcc` instruction is not an `n`, insert an `n` after the `j`. For example, `je` becomes `jne` and `jl` becomes `jnl`.
- If the second letter of the `jcc` instruction is an `n`, then remove that `n` from the instruction. For example, `jng` becomes `jg` and `jne` becomes `je`.

The two exceptions to this rule are `jpe` (jump if parity is even) and `jpo` (jump if parity is odd). These exceptions cause few problems because (1) you'll hardly ever need to test the parity flag, and (2) you can use the aliases `jp` and `jnp` as synonyms for `jpe` and `jpo`. The "N/No N" rule applies to `jp` and `jnp`.

Though you *know* that `jge` is the opposite of `jl`, get in the habit of using `jnl` rather than `jge` as the opposite jump instruction for `jl`. It's too easy in an important situation to start thinking "greater is the opposite of less" and substitute jg instead. You can avoid this confusion by always using the "N/No N" rule.

The 80x86 conditional jump instructions give you the ability to split program flow into one of two paths depending on some condition. Suppose you want to increment the AX register if BX is equal to CX. You can accomplish this with the following code:

```
          cmp( bx, cx );
          jne SkipStmts;
          inc( ax );
SkipStmts:
```

The trick is to use the *opposite* branch to skip over the instructions you want to execute if the condition is true. Always use the "opposite branch (N/No N)" rule given earlier to select the opposite branch.

You can also use the conditional jump instructions to synthesize loops. For example, the following code sequence reads a sequence of characters from the user and stores each character in successive elements of an array until the user presses the ENTER key (carriage return):

```
          mov( 0, edi );
RdLnLoop:
          stdin.getc();              // Read a character into the al register.
          mov( al, Input[ edi ] );   // Store away the character.
          inc( edi );                // Move on to the next character.
          cmp( al, stdio.cr );       // See if the user pressed Enter.
          jne RdLnLoop;
```

Like the set*cc* instructions, the conditional jump instructions come in two basic categories: those that test specific processor flags (e.g., jz, jc, jno) and those that test some condition (less than, greater than, etc.). When testing a condition, the conditional jump instructions almost always follow a cmp instruction. The cmp instruction sets the flags so that you can use a ja, jae, jb, jbe, je, or jne instruction to test for unsigned less than, less than or equal, equal, unequal, greater than, or greater than or equal. Simultaneously, the cmp instruction sets the flags so that you can also do a signed comparison using the jl, jle, je, jne, jg, and jge instructions.

The conditional jump instructions only test the 80x86 flags; they do not affect any of them.

7.5 "Medium-Level" Control Structures: jt and jf

HLA provides two special conditional jump instructions: jt (jump if true) and jf (jump if false). These instructions take the following syntax:

```
jt( boolean_expression ) target_label;
jf( boolean_expression ) target_label;
```

The `boolean_expression` is the standard HLA boolean expression allowed by `if..endif` and other HLA high-level language statements. These instructions evaluate the boolean expression and jump to the specified label if the expression evaluates true (`jt`) or false (`jf`).

These are not real 80x86 instructions. HLA compiles them into a sequence of one or more 80x86 machine instructions that achieve the same result. In general, you should not use these two instructions in your main code; they offer few benefits over using an `if..endif` statement and they are no more readable than the pure assembly language sequences they compile into. HLA provides these "medium-level" instructions so that you may create your own high-level control structures using macros (see Chapter 9 and the HLA reference manual for more details).

7.6 Implementing Common Control Structures in Assembly Language

Because a primary goal of this chapter is to teach you how to use the low-level machine instructions to implement decisions, loops, and other control constructs, it would be wise to show you how to implement these high-level statements using pure assembly language. The following sections provide this information.

7.7 Introduction to Decisions

In its most basic form, a *decision* is some sort of branch within the code that switches between two possible execution paths based on some condition. Normally (though not always), conditional instruction sequences are implemented with the conditional jump instructions. Conditional instructions correspond to the `if..then..endif` statement in HLA:

```
 if( expression ) then
     << statements >>
endif;
```

Assembly language, as usual, offers much more flexibility when dealing with conditional statements. Consider the following C/C++ statement:

```
if( (( x < y ) && ( z > t )) || ( a != b ) )
    stmt1;
```

A "brute force" approach to converting this statement into assembly language might produce the following:

```
    mov( x, eax );
    cmp( eax, y );
    setl( bl );        // Store x<y in bl.
    mov( z, eax );
    cmp( eax, t );
```

```
        setg( bh );       // Store z>t in bh.
        and( bh, bl );    // Put (x<y) && (z>t) into bl.
        mov( a, eax );
        cmp( eax, b );
        setne( bh );      // Store a != b into bh.
        or( bh, bl );     // Put (x<y) && (z>t) || (a!=b) into bl
        je SkipStmt1;     // Branch if result is false.

    << Code for Stmt1 goes here. >>

SkipStmt1:
```

As you can see, it takes a considerable number of conditional statements just to process the expression in the example above. This roughly corresponds to the (equivalent) C/C++ statements:

```
        bl = x < y;
        bh = z > t;
        bl = bl && bh;
        bh = a != b;
        bl = bl || bh;
        if( bl )
            << Stmt1 >>;
```

Now compare this with the following "improved" code:

```
        mov( a, eax );
        cmp( eax, b );
        jne DoStmt;
        mov( x, eax );
        cmp( eax, y );
        jnl SkipStmt;
        mov( z, eax );
        cmp( eax, t );
        jng SkipStmt;
DoStmt:
        << Place code for Stmt1 here. >>
SkipStmt:
```

Two things should be apparent from the code sequences above: First, a single conditional statement in C/C++ (or some other HLL) may require several conditional jumps in assembly language; second, organization of complex expressions in a conditional sequence can affect the efficiency of the code. Therefore, you should exercise care when dealing with conditional sequences in assembly language.

Conditional statements may be broken down into three basic categories: `if` statements, `switch`/`case` statements, and indirect jumps. The following sections describe these program structures, how to use them, and how to write them in assembly language.

7.7.1 if..then..else Sequences

The most common conditional statements are the `if..then..endif` and `if..then..else..endif` statements. These two statements take the form shown in Figure 7-1.

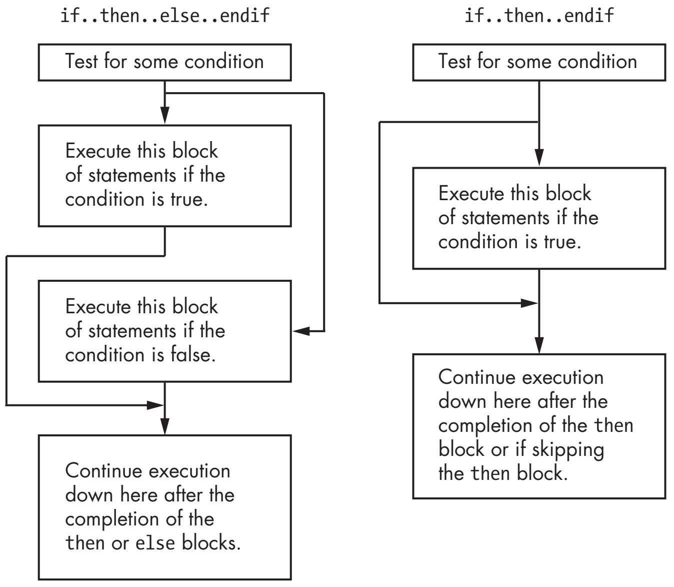

if..then..else..endif
Test for some condition
Execute this block of statements if the condition is true.
Execute this block of statements if the condition is false.
Continue execution down here after the completion of the then or else blocks.
if..then..endif
Test for some condition
Execute this block of statements if the condition is true.
Continue execution down here after the completion of the then block or if skipping the then block.

Figure 7-1: `if..then..else..endif` and `if..then..endif` statement flow

The `if..then..endif` statement is just a special case of the `if..then..else..endif` statement (with an empty `else` block). Therefore, we'll consider only the more general `if..then..else..endif` form. The basic implementation of an `if..then..else..endif` statement in 80x86 assembly language looks something like this:

```
    << Sequence of statements to test some condition >>
          jcc ElseCode;
    << Sequence of statements corresponding to the THEN block >>

          jmp EndOfIf;

ElseCode:
    << Sequence of statements corresponding to the ELSE block >>

EndOfIf:
```

Note that *jcc* represents some conditional jump instruction. For example, to convert the C/C++ statement

```
     if( a == b )
          c = d;
     else
          b = b + 1;
```

to assembly language, you could use the following 80x86 code:

```
          mov( a, eax );
          cmp( eax, b );
          jne ElsePart;
          mov( d, c );
          jmp EndOfIf;

ElseBlk:
          inc( b );

EndOfIf:
```

For simple expressions like `( a == b )` generating the proper code for an `if..then..else..endif` statement is almost trivial. Should the expression become more complex, the code complexity increases as well. Consider the following C/C++ `if` statement presented earlier:

```
     if( (( x > y ) && ( z < t )) || ( a != b ) )
          c = d;
```

When processing complex `if` statements such as this one, you'll find the conversion task easier if you break the `if` statement into a sequence of three different `if` statements as follows:

```
     if( a != b ) c = d;
     else if( x > y)
          if( z < t )
               c = d;
```

This conversion comes from the following C/C++ equivalents:

```
if( expr1 && expr2 ) stmt;
```

is equivalent to

```
if( expr1 ) if( expr2 ) stmt;
```

and

```
if( expr1 || expr2 ) stmt;
```

is equivalent to

```
if( expr1 ) stmt;
else if( expr2 ) stmt;
```

In assembly language, the former `if` statement becomes

```
// if( (( x > y ) && ( z < t )) || ( a != b ) )
//      c = d;

          mov( a, eax );
          cmp( eax, b );
          jne DoIF;
          mov( x, eax );
          cmp( eax, y );
          jng EndOfIF;
          mov( z, eax );
          cmp( eax, t );
          jnl EndOfIf;
DoIf:
          mov( d, eax );
          mov( eax, c );
EndOfIf:
```

As you can see, testing a condition can easily become more complex than the statements appearing in the `else` and `then` blocks. Although it seems somewhat paradoxical that it may take more effort to test a condition than to act on the results of that condition, it happens all the time. Therefore, you should be prepared to accept this.

Probably the biggest problem with complex conditional statements in assembly language is trying to figure out what you've done after you've written the code. A big advantage high-level languages offer over assembly language is that expressions are much easier to read and comprehend. The high-level version is (more) self-documenting, whereas assembly language tends to hide the true nature of the code. Therefore, well-written comments are an essential ingredient to assembly language implementations of `if..then..else..endif` statements. An elegant implementation of the example above is as follows:

```
// if ((x > y) && (z < t)) or (a != b)  c = d;
// Implemented as:
// if (a != b) then goto DoIf;

          mov( a, eax );
          cmp( eax, b );
          jne DoIf;

// if not (x > t) then goto EndOfIf;

          mov( x, eax );
          cmp( eax, y );
          jng EndOfIf;

// if not (z < t) then goto EndOfIf;
```

```
          mov( z, eax );
          cmp( eax, t );
          jnl EndOfIf;

// then block:

DoIf:
          mov( d, eax );
          mov( eax, c );

// End of if statement

EndOfIf:
```

Admittedly, this appears to be going overboard for such a simple example. The following would probably suffice:

```
// if ( (( x > y ) && ( z < t )) || ( a != b ) )  c = d;
// Test the boolean expression:

          mov( a, eax );
          cmp( eax, b );
          jne DoIf;
          mov( x, eax );
          cmp( eax, y );
          jng EndOfIf;
          mov( z, eax );
          cmp( eax, t );
          jnl EndOfIf;

// then block:

DoIf:
          mov( d, eax );
          mov( eax, c );

// End of if statement

EndOfIf:
```

However, as your `if` statements become complex, the density (and quality) of your comments become more and more important.

7.7.2 Translating HLA if Statements into Pure Assembly Language

Translating HLA `if` statements into pure assembly language is very easy. The boolean expressions that the HLA `if` statement supports were specifically chosen to expand into a few simple machine instructions. The following paragraphs discuss the conversion of each supported boolean expression into pure machine code.

if(*flag_specification*) then *stmts* endif;

This form is, perhaps, the easiest HLA `if` statement to convert. To execute the code immediately following the `then` keyword if a particular flag is set (or clear), all you need do is skip over the code if the flag is clear (set). This requires only a single conditional jump instruction for implementation, as the following examples demonstrate:

```
// if( @c ) then inc( eax );  endif;

          jnc SkipTheInc;

               inc( eax );

          SkipTheInc:

// if( @ns ) then neg( eax ); endif;

          js SkipTheNeg;

               neg( eax );

          SkipTheNeg:
```

if(*register*) then *stmts* endif;

This form uses the `test` instruction to check the specified register for 0. If the register contains 0 (false), then the program jumps around the statements after the `then` clause with a `jz` instruction. Converting this statement to assembly language requires a `test` instruction and a `jz` instruction, as the following examples demonstrate:

```
// if( eax ) then mov( false, eax );  endif;

          test( eax, eax );
          jz DontSetFalse;

               mov( false, eax );

          DontSetFalse:

// if( al ) then mov( bl, cl );  endif;

          test( al, al );
          jz noMove;

               mov( bl, cl );

          noMove:
```

if(!*register*) then *stmts* endif;

This form of the if statement uses the test instruction to check the specified register to see if it is 0. If the register is not 0 (true), then the program jumps around the statements after the then clause with a jnz instruction. Converting this statement to assembly language requires a test instruction and a jnz instruction in a manner identical to the previous examples.

if(*boolean_variable*) then *stmts* endif;

This form of the if statement compares the boolean variable against 0 (false) and branches around the statements if the variable contains false. HLA implements this statement by using the cmp instruction to compare the boolean variable to 0, and then it uses a jz (je) instruction to jump around the statements if the variable is false. The following example demonstrates the conversion:

```
// if( bool ) then mov( 0, al );  endif;

          cmp( bool, false );
          je SkipZeroAL;

               mov( 0, al );

          SkipZeroAL:
```

if(!*boolean_variable*) then *stmts* endif;

This form of the if statement compares the boolean variable against 0 (false) and branches around the statements if the variable contains true (the opposite condition of the previous example). HLA implements this statement by using the cmp instruction to compare the boolean variable to 0 and then it uses a jnz (jne) instruction to jump around the statements if the variable contains true. The following example demonstrates the conversion:

```
// if( !bool ) then mov( 0, al );  endif;

          cmp( bool, false );
          jne SkipZeroAL;

               mov( 0, al );

          SkipZeroAL:
```

if(*mem_reg relop mem_reg_const*) then *stmts* endif;

HLA translates this form of the if statement into a cmp instruction and a conditional jump that skips over the statements on the opposite condition specified by the relop operator. Table 7-4 lists the correspondence between operators and conditional jump instructions.

Table 7-4: `if` Statement Conditional Jump Instructions

Relational operation	Conditional jump instruction if both operands are unsigned	Conditional jump instruction if either operand is signed
= or ==	jne	jne
<> or !=	je	je
<	jnb	jnl
<=	jnbe	jnle
>	jna	jng
>=	jnae	jnge

Here are a few examples of `if` statements translated into pure assembly language that use expressions involving relational operators:

```
// if( al == ch ) then inc( cl ); endif;

          cmp( al, ch );
          jne SkipIncCL;

               inc( cl );

          SkipIncCL:

// if( ch >= 'a' ) then and( $5f, ch ); endif;

          cmp( ch, 'a' );
          jnae NotLowerCase

               and( $5f, ch );

          NotLowerCase:

// if( (type int32 eax ) < -5 ) then mov( -5, eax );  endif;

          cmp( eax, -5 );
          jnl DontClipEAX;

               mov( -5, eax );

          DontClipEAX:

// if( si <> di ) then inc( si );  endif;

          cmp( si, di );
          je DontIncSI;

               inc( si );

          DontIncSI:
```

if(*reg/mem* in *LowConst..HiConst*) then *stmts* endif;

HLA translates this `if` statement into a pair of `cmp` instructions and a pair of conditional jump instructions. It compares the register or memory location against the lower-valued constant and jumps if less than (signed) or below (unsigned) past the statements after the `then` clause. If the register or memory location's value is greater than or equal to *`LowConst`*, the code falls through to the second `cmp` and conditional jump pair that compares the register or memory location against the higher constant. If the value is greater than (above) this constant, a conditional jump instruction skips the statements in the `then` clause.

Here's an example:

```
// if( eax in 1000..125_000 ) then sub( 1000, eax );  endif;

          cmp( eax, 1000 );
          jb DontSub1000;
          cmp( eax, 125_000 );
          ja DontSub1000;

               sub( 1000, eax );

          DontSub1000:

// if( i32 in -5..5 ) then add( 5, i32 ); endif;

          cmp( i32, -5 );
          jl NoAdd5;
          cmp( i32, 5 );
          jg NoAdd5;

               add(5, i32 );

          NoAdd5:
```

if(*reg/mem* not in *LowConst..HiConst*) then *stmts* endif;

This form of the HLA `if` statement tests a register or memory location to see if its value is outside a specified range. The implementation is very similar to the previous code except you branch to the `then` clause if the value is less than the *`LowConst`* value or greater than the *`HiConst`* value, and you branch over the code in the `then` clause if the value is within the range specified by the two constants. The following examples demonstrate how to do this conversion:

```
// if( eax not in 1000..125_000 ) then add( 1000, eax );  endif;

          cmp( eax, 1000 );
          jb Add1000;
          cmp( eax, 125_000 );
          jbe SkipAdd1000;
```

```
            Add1000:
            add( 1000, eax );

        SkipAdd1000:

// if( i32 not in -5..5 ) then mov( 0, i32 );  endif;

        cmp( i32, -5 );
        jl Zeroi32;
        cmp( i32, 5 );
        jle SkipZero;

            Zeroi32:
            mov( 0, i32 );

        SkipZero:
```

7.7.3 Implementing Complex if Statements Using Complete Boolean Evaluation

Many boolean expressions involve conjunction (`and`) or disjunction (`or`) operations. This section describes how to convert boolean expressions into assembly language. There are two different ways to convert complex boolean expressions involving conjunction and disjunction into assembly language: using complete boolean evaluation or using short-circuit boolean evaluation. This section discusses complete boolean evaluation. The next section discusses short-circuit boolean evaluation.

Conversion via complete boolean evaluation is almost identical to converting arithmetic expressions into assembly language. Indeed, the previous chapter on arithmetic covers this conversion process. About the only thing worth noting about that process is that you do not need to store the result in some variable; once the evaluation of the expression is complete, you check to see if you have a false (0) or true (1, or nonzero) result to take whatever action the boolean expression dictates. As you can see in the examples in the preceding sections, you can often use the fact that the last logical instruction (`and`/`or`) sets the zero flag if the result is false and clears the zero flag if the result is true. This lets you avoid explicitly testing for the result. Consider the following `if` statement and its conversion to assembly language using complete boolean evaluation:

```
//     if( (( x < y ) && ( z > t )) || ( a != b ) )
//         << Stmt1 >>;

        mov( x, eax );
        cmp( eax, y );
        setl( bl );     // Store x<y in bl.
        mov( z, eax );
        cmp( eax, t );
        setg( bh );     // Store z>t in bh.
        and( bh, bl );  // Put (x<y) && (z>t) into bl.
        mov( a, eax );
        cmp( eax, b );
        setne( bh );    // Store a != b into bh.
        or( bh, bl );   // Put (x<y) && (z>t) || (a != b) into bl.
        je SkipStmt1;   // Branch if result is false.

    << Code for Stmt1 goes here. >>

SkipStmt1:
```

This code computes a boolean result in the BL register and then, at the end of the computation, tests this value to see if it contains true or false. If the result is false, this sequence skips over the code associated with `Stmt1`. The important thing to note in this example is that the program will execute each and every instruction that computes this boolean result (up to the `je` instruction).

7.7.4 Short-Circuit Boolean Evaluation

If you are willing to expend a little more effort, you can usually convert a boolean expression to a much shorter and faster sequence of assembly language instructions using *short-circuit boolean evaluation*. Short-circuit boolean evaluation attempts to determine whether an expression is true or false by executing only some of the instructions that would compute the complete expression. For this reason, plus the fact that short-circuit boolean evaluation doesn't require the use of any temporary registers, HLA uses short-circuit evaluation when translating complex boolean expressions into assembly language.

Consider the expression `a && b`. Once we determine that `a` is false, there is no need to evaluate `b` because there is no way the expression can be true. If and `b` represent subexpressions rather than simple variables, the savings possible with short-circuit boolean evaluation are apparent. As a concrete example, consider the subexpression `((x<y) && (z>t))` from the previous section. Once you determine that `x` is not less than `y`, there is no need to check to see if `z` is greater than `t` because the expression will be false regardless of `z` and `t`'s values. The following code fragment shows how you can implement short-circuit boolean evaluation for this expression:

```
// if( (x<y) && (z>t) ) then ...

        mov( x, eax );
        cmp( eax, y );
        jnl TestFails;
        mov( z, eax );
        cmp( eax, t );
        jng TestFails;

            << Code for THEN clause of IF statement >>

        TestFails:
```

Notice how the code skips any further testing once it determines that x is not less than y. Of course, if x is less than y, then the program has to test z to see if it is greater than t; if not, the program skips over the then clause. Only if the program satisfies both conditions does the code fall through to the then clause.

For the logical or operation the technique is similar. If the first subexpression evaluates to true, then there is no need to test the second operand. Whatever the second operand's value is at that point, the full expression still evaluates to true. The following example demonstrates the use of short-circuit evaluation with disjunction (or):

```
// if( ch < 'A' || ch > 'Z' )
//     then stdout.put( "Not an uppercase char" );
// endif;

          cmp( ch, 'A' );
          jb ItsNotUC
          cmp( ch, 'Z' );
          jna ItWasUC;

               ItsNotUC:
               stdout.put( "Not an uppercase char" );

          ItWasUC:
```

Because the conjunction and disjunction operators are commutative, you can evaluate the left or right operand first if it is more convenient to do so.[2] As one last example in this section, consider the full boolean expression from the previous section:

```
// if( (( x < y ) && ( z > t )) || ( a != b ) ) << Stmt1 >>;

          mov( a, eax );
          cmp( eax, b );
          jne DoStmt1;
          mov( x, eax );
          cmp( eax, y );
          jnl SkipStmt1;
          mov( z, eax );
          cmp( eax, t );
jng SkipStmt1;

               DoStmt1:
               << Code for Stmt1 goes here. >>

          SkipStmt1:
```

[2] However, be aware of the fact that some expressions depend on the leftmost subexpression evaluating one way in order for the rightmost subexpression to be valid; for example, a common test in C/C++ is if(x != NULL && x->y)...

Notice how the code in this example chose to evaluate `a != b` first and the remaining subexpression last. This is a common technique assembly language programmers use to write better code.

7.7.5 Short-Circuit vs. Complete Boolean Evaluation

When using complete boolean evaluation, every statement in the sequence for that expression will execute; short-circuit boolean evaluation, on the other hand, may not require the execution of every statement associated with the boolean expression. As you've seen in the previous two sections, code based on short-circuit evaluation is usually shorter and faster. So it would seem that short-circuit evaluation is the technique of choice when converting complex boolean expressions to assembly language.

Sometimes, unfortunately, short-circuit boolean evaluation may not produce the correct result. In the presence of *side effects* in an expression, short-circuit boolean evaluation will produce a different result than complete boolean evaluation. Consider the following C/C++ example:

```
        if( ( x == y ) && ( ++z != 0 )) << Stmt >>;
```

Using complete boolean evaluation, you might generate the following code:

```
        mov( x, eax );      // See if x == y.
        cmp( eax, y );
        sete( bl );
        inc( z );           // ++z
        cmp( z, 0 );        // See if incremented z is 0.
        setne( bh );
        and( bh, bl );      // Test x == y && ++z != 0.
        jz SkipStmt;

        << Code for Stmt goes here. >>

SkipStmt:
```

Using short-circuit boolean evaluation, you might generate the following code:

```
        mov( x, eax );      // See if x == y.
        cmp( eax, y );
        jne SkipStmt;
        inc( z );           // ++z
        cmp( z, 0 );        // See if incremented z is 0.
        je SkipStmt;

        << Code for Stmt goes here. >>

SkipStmt:
```

Notice a very subtle but important difference between these two conversions: If x is equal to y, then the first version above *still increments* z and compares it to 0 before it executes the code associated with Stmt; the short-circuit version, on the other hand, skips the code that increments z if it turns out that x is equal to y. Therefore, the behavior of these two code fragments is different if x is equal to y. Neither implementation is particularly wrong; depending on the circumstances you may or may not want the code to increment z if x is equal to y. However, it is important that you realize that these two schemes produce different results, so you can choose an appropriate implementation if the effect of this code on z matters to your program.

Many programs take advantage of short-circuit boolean evaluation and rely on the fact that the program may not evaluate certain components of the expression. The following C/C++ code fragment demonstrates what is probably the most common example that requires short-circuit boolean evaluation:

```
                    if( Ptr != NULL && *Ptr == 'a' ) << Stmt >>;
```

If it turns out that Ptr is NULL, then the expression is false and there is no need to evaluate the remainder of the expression (and, therefore, code that uses short-circuit boolean evaluation will not evaluate the remainder of this expression). This statement relies on the semantics of short-circuit boolean evaluation for correct operation. Were C/C++ to use complete boolean evaluation, and the variable Ptr contained NULL, then the second half of the expression would attempt to dereference a NULL pointer (which tends to crash most programs). Consider the translation of this statement using complete and short-circuit boolean evaluation:

```
// Complete boolean evaluation:

          mov( Ptr, eax );
          test( eax, eax );    // Check to see if eax is 0 (NULL is 0).
          setne( bl );
          mov( [eax], al );    // Get *Ptr into al.
          cmp( al, 'a' );
          sete( bh );
          and( bh, bl );
          jz SkipStmt;

          << Code for Stmt goes here. >>

SkipStmt:
```

Notice in this example that if Ptr contains NULL (0), then this program will attempt to access the data at location 0 in memory via the mov([eax], al); instruction. Under most operating systems this will cause a memory access fault (general protection fault).

Now consider the short-circuit boolean conversion:

```
// Short-circuit boolean evaluation

          mov( Ptr, eax );     // See if Ptr contains NULL (0) and
          test( eax, eax );    // immediately skip past Stmt if this
          jz SkipStmt;         // is the case.

          mov( [eax], al );    // If we get to this point, Ptr contains
          cmp( al, 'a' );      // a non-NULL value, so see if it points
          jne SkipStmt;        // at the character 'a'.

          << Code for Stmt goes here. >>

SkipStmt:
```

As you can see in this example, the problem with dereferencing the NULL pointer doesn't exist. If Ptr contains NULL, this code skips over the statements that attempt to access the memory address Ptr contains.

7.7.6 Efficient Implementation of if Statements in Assembly Language

Encoding if statements efficiently in assembly language takes a bit more thought than simply choosing short-circuit evaluation over complete boolean evaluation. To write code that executes as quickly as possible in assembly language, you must carefully analyze the situation and generate the code appropriately. The following paragraphs provide some suggestions you can apply to your programs to improve their performance.

7.7.6.1 Know Your Data!

A mistake programmers often make is the assumption that data is random. In reality, data is rarely random, and if you know the types of values that your program commonly uses, you can use this knowledge to write better code. To see how, consider the following C/C++ statement:

```
          if(( a == b ) && ( c < d )) ++i;
```

Because C/C++ uses short-circuit evaluation, this code will test to see if a is equal to b. If so, then it will test to see if c is less than d. If you expect a to be equal to b most of the time but don't expect c to be less than d most of the time, this statement will execute slower than it should. Consider the following HLA implementation of this code:

```
          mov( a, eax );
          cmp( eax, b );
          jne DontIncI;
```

```
        mov( c, eax );
        cmp( eax, d );
        jnl DontIncI;

            inc( i );

        DontIncI:
```

As you can see in this code, if `a` is equal to `b` most of the time and `c` is not less than `d` most of the time, you will have to execute all six instructions nearly every time in order to determine that the expression is false. Now consider the following implementation of the above C/C++ statement that takes advantage of this knowledge and the fact that the `&&` operator is commutative:

```
        mov( c, eax );
        cmp( eax, d );
        jnl DontIncI;

        mov( a, eax );
        cmp( eax, b );
        jne DontIncI;

            inc( i );

        DontIncI:
```

In this example the code first checks to see if `c` is less than `d`. If most of the time `c` is less than `d`, then this code determines that it has to skip to the label `DontIncI` after executing only three instructions in the typical case (compared with six instructions in the previous example). This fact is much more obvious in assembly language than in a high-level language; this is one of the main reasons why assembly programs are often faster than their high-level language counterparts: optimizations are more obvious in assembly language than in a high-level language. Of course, the key here is to understand the behavior of your data so you can make intelligent decisions such as the one above.

7.7.6.2 Rearranging Expressions

Even if your data is random (or you can't determine how the input values will affect your decisions), there may still be some benefit to rearranging the terms in your expressions. Some calculations take far longer to compute than others. For example, the `div` instruction is much slower than a simple `cmp` instruction. Therefore, if you have a statement like the following, you may want to rearrange the expression so that the `cmp` comes first:

```
if( (x % 10 = 0 ) && (x != y ) ++x;
```

Converted to assembly code, this if statement becomes:

```
        mov( x, eax );           // Compute X % 10.
        cdq();                   // Must sign extend eax -> edx:eax.
        imod( 10, edx:eax );     // Remember, remainder goes into edx.
        test( edx, edx );        // See if edx is 0.
        jnz SkipIf;

        mov( x, eax );
        cmp( eax, y );
        je SkipIf;

            inc( x );

        SkipIf:
```

The imod instruction is very expensive (often 50–100 times slower than most of the other instructions in this example). Unless it is 50–100 times more likely that the remainder is 0 rather than x is equal to y, it would be better to do the comparison first and the remainder calculation afterward:

```
        mov( x, eax );
        cmp( eax, y );
        je SkipIf;

        mov( x, eax );           // Compute X % 10.
        cdq();                   // Must sign extend eax -> edx:eax.
        imod( 10, edx:eax );     // Remember, remainder goes into edx.
        test( edx, edx );        // See if edx is 0.
        jnz SkipIf;

            inc( x );

        SkipIf:
```

Of course, in order to rearrange the expression in this manner, the code must not assume the use of short-circuit evaluation semantics (because the && and || operators are not commutative if the code must compute one subexpression before another).

7.7.6.3 Destructuring Your Code

Although there are many good things to be said about structured programming techniques, there are some drawbacks to writing structured code. Specifically, structured code is sometimes less efficient than unstructured code. Most of the time this is tolerable because unstructured code is difficult to read and maintain; it is often acceptable to sacrifice some performance in exchange for maintainable code. In certain instances, however, you may need all the performance you can get. In those rare instances you might choose to compromise the readability of your code in order to gain some additional performance.

One classic way to do this is to use code movement to move code your program rarely uses out of the way of code that executes most of the time. For example, consider the following pseudo C/C++ statement:

```
if( See_If_an_Error_Has_Occurred )
{
     << Statements to execute if no error >>
}
else
{
     << Error handling statements >>
}
```

In normal code, one does not expect errors to be frequent. Therefore, you would normally expect the `then` section of the above `if` to execute far more often than the `else` clause. The code above could translate into the following assembly code:

```
cmp( See_If_an_Error_Has_Occurred, true );
je HandleTheError;

     << Statements to execute if no error >>
     jmp EndOfIF;

HandleTheError:
     << Error handling statements >>
EndOfIf:
```

Notice that if the expression is false, this code falls through to the normal statements and then jumps over the error-handling statements. Instructions that transfer control from one point in your program to another (for example, `jmp` instructions) tend to be slow. It is much faster to execute a sequential set of instructions rather than jump all over the place in your program. Unfortunately, the code above doesn't allow this. One way to rectify this problem is to move the `else` clause of the code somewhere else in your program. That is, you could rewrite the code as follows:

```
cmp( See_If_an_Error_Has_Occurred, true );
je HandleTheError;

     << Statements to execute if no error >>

EndOfIf:
```

At some other point in your program (typically after a `jmp` instruction) you would insert the following code:

```
HandleTheError:
     << Error handling statements >>
     jmp EndOfIf;
```

Note that the program isn't any shorter. The jmp you removed from the original sequence winds up at the end of the else clause. However, because the else clause rarely executes, moving the jmp instruction from the then clause (which executes frequently) to the else clause is a big performance win because the then clause executes using only straight-line code. This technique is surprisingly effective in many time-critical code segments.

There is a difference between writing *destructured* code and writing *unstructured* code. Unstructured code is written in an unstructured way to begin with. It is generally hard to read, difficult to maintain, and often contains defects. Destructured code, on the other hand, starts out as structured code, and you make a conscious decision to eliminate the structure in order to gain a small performance boost. Generally, you've already tested the code in its structured form before destructuring it. Therefore, destructured code is often easier to work with than unstructured code.

7.7.6.4 Calculation Rather Than Branching

On many processors in the 80x86 family, branches (jumps) are very expensive compared to many other instructions. For this reason it is sometimes better to execute more instructions in a sequence than fewer instructions that involve branching. For example, consider the simple assignment eax = abs(eax);. Unfortunately, there is no 80x86 instruction that computes the absolute value of an integer. The obvious way to handle this is with an instruction sequence like the following:

```
          test( eax, eax );
          jns ItsPositive;

               neg( eax );

          ItsPositive:
```

However, as you can plainly see in this example, it uses a conditional jump to skip over the neg instruction (that creates a positive value in EAX if EAX was negative). Now consider the following sequence that will also do the job:

```
// Set edx to $FFFF_FFFF if eax is negative, $0000_0000 if eax is
// 0 or positive:

          cdq();

// If eax was negative, the following code inverts all the bits in eax;
// otherwise it has no effect on eax.

          xor( edx, eax );

// If eax was negative, the following code adds 1 to eax; otherwise
// it doesn't modify eax's value.
```

```
        and( 1, edx );      // edx = 0 or 1 (1 if eax was negative).
        add( edx, eax );
```

This code will invert all the bits in EAX and then add 1 to EAX if EAX was negative prior to the sequence; that is, it negates the value in EAX. If EAX was 0 or positive, then this code does not change the value in EAX.

Note that this sequence takes four instructions rather than the three the previous example requires. However, because there are no transfer-of-control instructions in this sequence, it may execute faster on many CPUs in the 80x86 family.

7.7.7 switch/case Statements

The HLA `switch` statement takes the following form:

```
     switch( reg32 )
         case( const1 )
              << Stmts1: code to execute if reg32 equals const1 >>

         case( const2 )
              << Stmts2: code to execute if reg32 equals const2 >>
           .
           .
           .
         case( constn )
              << Stmtsn: code to execute if reg32 equals constn >>

         default      // Note that the default section is optional.
              << Stmts_default: code to execute if reg32
                 does not equal any of the case values >>

    endswitch;
```

When this statement executes, it checks the value of the register against the constants *`const1..constn`*. If a match is found, then the corresponding statements execute. HLA places a few restrictions on the `switch` statement. First, the HLA `switch` statement allows only a 32-bit register as the `switch` expression. Second, all the constants in the `case` clauses must be unique. The reason for these restrictions will become clear in a moment.

Most introductory programming texts introduce the `switch/case` statement by explaining it as a sequence of `if..then..elseif..else..endif` statements. They might claim that the following two pieces of HLA code are equivalent:

```
    switch( eax )
         case(0) stdout.put("i=0");
         case(1) stdout.put("i=1");
         case(2) stdout.put("i=2");
    endswitch;
```

```
     if( eax = 0 ) then
          stdout.put("i=0")
     elseif( eax = 1 ) then
          stdout.put("i=1")
     elseif( eax = 2 ) then
          stdout.put("i=2");
     endif;
```

While semantically these two code segments may be the same, their implementation is usually different. Whereas the `if..then..elseif..else..endif` chain does a comparison for each conditional statement in the sequence, the `switch` statement normally uses an indirect jump to transfer control to any one of several statements with a single computation. Consider the two examples presented above; they could be written in assembly language with the following code:

```
// if..then..else..endif form:

          mov( i, eax );
          test( eax, eax );   // Check for 0.
          jnz Not0;
               stdout.put( "i=0" );
               jmp EndCase;

          Not0:
          cmp( eax, 1 );
          jne Not1;
               stdou.put( "i=1" );
               jmp EndCase;

          Not1:
          cmp( eax, 2 );
          jne EndCase;
               stdout.put( "i=2" );
     EndCase:


// Indirect Jump Version

readonly
     JmpTbl:dword[3] := [ &Stmt0, &Stmt1, &Stmt2 ];
           .
           .
           .
     mov( i, eax );
     jmp( JmpTbl[ eax*4 ] );

          Stmt0:
               stdout.put( "i=0" );
               jmp EndCase;
```

```
        Stmt1:
            stdout.put( "I=1" );
            jmp EndCase;

        Stmt2:
            stdout.put( "I=2" );

    EndCase:
```

The implementation of the `if..then..elseif..else..endif` version is fairly obvious and needs little in the way of explanation. The indirect jump version, however, is probably quite mysterious to you, so let's consider how this particular implementation of the `switch` statement works.

Remember that there are three common forms of the `jmp` instruction. The standard unconditional `jmp` instruction, like the `jmp EndCase;` instruction in the previous examples, transfers control directly to the statement label specified as the `jmp` operand. The second form of the `jmp` instruction—`jmp(` *`reg32`* `);`—transfers control to the memory location specified by the address found in a 32-bit register. The third form of the `jmp` instruction, the one the previous example uses, transfers control to the instruction specified by the contents of a double-word memory location. As this example clearly illustrates, that memory location can use any addressing mode. You are not limited to the displacement-only addressing mode. Now let's consider exactly how this second implementation of the `switch` statement works.

To begin with, a `switch` statement requires that you create an array of pointers with each element containing the address of a statement label in your code (those labels must be attached to the sequence of instructions to execute for each case in the `switch` statement). In the example above, the `JmpTbl` array serves this purpose. Note that this code initializes `JmpTbl` with the address of the statement labels `Stmt0`, `Stmt1`, and `Stmt2`. The program places this array in the `readonly` section because the program should never change these values during execution.

WARNING *Whenever you initialize an array with a set of addresses of statement labels as in this example, the declaration section in which you declare the array (e.g.,* `readonly` *in this case) must be in the same procedure that contains the statement labels.*[3]

During the execution of this code sequence, the program loads the EAX register with `i`'s value. Then the program uses this value as an index into the `JmpTbl` array and transfers control to the 4-byte address found at the specified location. For example, if EAX contains 0, the `jmp( JmpTbl[eax*4] );` instruction will fetch the double word at address `JmpTbl+0 ( eax*4=0 )`. Because the first double word in the table contains the address of `Stmt0`, the `jmp` instruction transfers control to the first instruction following the `Stmt0` label. Likewise, if `i` (and therefore, EAX) contains 1, then the indirect `jmp` instruction fetches the double word at offset 4 from the table and transfers control to the first instruction following the `Stmt1` label (because the address of `Stmt1` appears at offset

[3] If the `switch` statement appears in your main program, you must declare the array in the declaration section of your main program.

4 in the table). Finally, if i/EAX contains 2, then this code fragment transfers control to the statements following the Stmt2 label because it appears at offset 8 in the JmpTbl table.

You should note that as you add more (consecutive) cases, the jump table implementation becomes more efficient (in terms of both space and speed) than the if/elseif form. Except for simple cases, the switch statement is almost always faster and usually by a large margin. As long as the case values are consecutive, the switch statement version is usually smaller as well.

What happens if you need to include nonconsecutive case labels or you cannot be sure that the switch value doesn't go out of range? With the HLA switch statement, such an occurrence will transfer control to the first statement after the endswitch clause (or to a default case, if one is present in the switch). However, this doesn't happen in the example above. If variable i does not contain 0, 1, or 2, executing the code above produces undefined results. For example, if i contains 5 when you execute the code in the previous example, the indirect jmp instruction will fetch the dword at offset 20 (5 * 4) in JmpTbl and transfer control to that address. Unfortunately, JmpTbl doesn't have six entries; so the program will wind up fetching the value of the third double word following JmpTbl and use that as the target address. This will often crash your program or transfer control to an unexpected location.

The solution is to place a few instructions before the indirect jmp to verify that the switch selection value is within some reasonable range. In the previous example, we'd probably want to verify that i's value is in the range 0..2 before executing the jmp instruction. If i's value is outside this range, the program should simply jump to the endcase label (this corresponds to dropping down to the first statement after the endswitch clause). The following code provides this modification:

```
readonly
    JmpTbl:dword[3] := [ &Stmt0, &Stmt1, &Stmt2 ];
     .
     .
     .
    mov( i, eax );
    cmp( eax, 2 );          // Verify that i is in the range
    ja EndCase;             // 0..2 before the indirect jmp.
    jmp( JmpTbl[ eax*4 ] );


        Stmt0:
            stdout.put( "i=0" );
            jmp EndCase;

        Stmt1:
            stdout.put( "i=1" );
            jmp EndCase;
```

```
          Stmt2:
               stdout.put( "i=2" );

     EndCase:
```

Although the example above handles the problem of selection values being outside the range 0..2, it still suffers from a couple of severe restrictions:

- The cases must start with the value 0. That is, the minimum case constant has to be 0 in this example.
- The case values must be contiguous.

Solving the first problem is easy, and you deal with it in two steps. First, you must compare the case selection value against a lower and upper bounds before determining if the case value is legal. For example:

```
// SWITCH statement specifying cases 5, 6, and 7:
// WARNING: This code does *NOT* work. Keep reading to find out why.

     mov( i, eax );
     cmp( eax, 5 );
     jb EndCase
     cmp( eax, 7 );              // Verify that i is in the range
     ja EndCase;                 // 5..7 before the indirect jmp.
     jmp( JmpTbl[ eax*4 ] );


          Stmt5:
               stdout.put( "i=5" );
               jmp EndCase;

          Stmt6:
               stdout.put( "i=6" );
               jmp EndCase;

          Stmt7:
               stdout.put( "i=7" );

     EndCase:
```

As you can see, this code adds a pair of extra instructions, `cmp` and `jb`, to test the selection value to ensure it is in the range 5..7. If not, control drops down to the `EndCase` label; otherwise control transfers via the indirect `jmp` instruction. Unfortunately, as the comments point out, this code is broken. Consider what happens if variable `i` contains the value 5: the code will verify that 5 is in the range 5..7 and then it will fetch the dword at offset 20 (`5*@size(dword)`) and jump to that address. As before, however, this loads 4 bytes outside the bounds of the table and does not transfer control to a defined location. One solution is to subtract the smallest case selection value from EAX before executing the `jmp` instruction, as shown in the following example.

```
// SWITCH statement specifying cases 5, 6, and 7:
// WARNING: There is a better way to do this. Keep reading.

readonly
     JmpTbl:dword[3] := [ &Stmt5, &Stmt6, &Stmt7 ];
           .
           .
           .
     mov( i, eax );
     cmp( eax, 5 );
     jb EndCase
     cmp( eax, 7 );              // Verify that i is in the range
     ja EndCase;                 // 5..7 before the indirect jmp.
     sub( 5, eax );              // 5->0, 6->1, 7->2.
     jmp( JmpTbl[ eax*4 ] );


          Stmt5:
               stdout.put( "i=5" );
               jmp EndCase;

          Stmt6:
               stdout.put( "i=6" );
               jmp EndCase;

          Stmt7:
               stdout.put( "i=7" );

     EndCase:
```

By subtracting 5 from the value in EAX, this code forces EAX to take on the value 0, 1, or 2 prior to the `jmp` instruction. Therefore, case-selection value 5 jumps to `Stmt5`, case-selection value 6 transfers control to `Stmt6`, and case-selection value 7 jumps to `Stmt7`.

There is a sneaky way to improve the code above. You can eliminate the `sub` instruction by merging this subtraction into the `jmp` instruction's address expression. Consider the following code that does this:

```
// SWITCH statement specifying cases 5, 6, and 7:

readonly
     JmpTbl:dword[3] := [ &Stmt5, &Stmt6, &Stmt7 ];
           .
           .
           .
     mov( i, eax );
     cmp( eax, 5 );
     jb EndCase
     cmp( eax, 7 );              // Verify that i is in the range
     ja EndCase;                 // 5..7 before the indirect jmp.
     jmp( JmpTbl[ eax*4 - 5*@size(dword)] );
```

```
        Stmt5:
            stdout.put( "i=5" );
            jmp EndCase;

        Stmt6:
            stdout.put( "i=6" );
            jmp EndCase;

        Stmt7:
            stdout.put( "i=7" );

    EndCase:
```

The HLA `switch` statement provides a `default` clause that executes if the case-selection value doesn't match any of the case values. For example:

```
    switch( ebx )

        case( 5 )  stdout.put( "ebx=5" );
        case( 6 )  stdout.put( "ebx=6" );
        case( 7 )  stdout.put( "ebx=7" );
        default
            stdout.put( "ebx does not equal 5, 6, or 7" );

    endswitch;
```

Implementing the equivalent of the `default` clause in pure assembly language is very easy. Just use a different target label in the `jb` and `ja` instructions at the beginning of the code. The following example implements an HLA `switch` statement similar to the one immediately above:

```
// SWITCH statement specifying cases 5, 6, and 7 with a DEFAULT clause:

readonly
    JmpTbl:dword[3] := [ &Stmt5, &Stmt6, &Stmt7 ];
          .
          .
          .
    mov( i, eax );
    cmp( eax, 5 );
    jb DefaultCase;
    cmp( eax, 7 );             // Verify that i is in the range
    ja DefaultCase;            // 5..7 before the indirect jmp.
    jmp( JmpTbl[ eax*4 - 5*@size(dword)] );


        Stmt5:
            stdout.put( "i=5" );
            jmp EndCase;
          Stmt6:
               stdout.put( "i=6" );
               jmp EndCase;

          Stmt7:
               stdout.put( "i=7" );
               jmp EndCase;

          DefaultCase:
               stdout.put( "i does not equal 5, 6, or 7" );
     EndCase:
```

The second restriction noted earlier, that the case values need to be contiguous, is easy to handle by inserting extra entries into the jump table. Consider the following HLA `switch` statement:

```
     switch( ebx )

          case( 1 ) stdout.put( "ebx = 1" );
          case( 2 ) stdout.put( "ebx = 2" );
          case( 4 ) stdout.put( "ebx = 4" );
          case( 8 ) stdout.put( "ebx = 8" );
          default
               stdout.put( "ebx is not 1, 2, 4, or 8" );

     endswitch;
```

The minimum switch value is 1 and the maximum value is 8. Therefore, the code before the indirect `jmp` instruction needs to compare the value in EBX against 1 and 8. If the value is between 1 and 8, it's still possible that EBX might not contain a legal case-selection value. However, because the `jmp` instruction indexes into a table of double words using the case-selection table, the table must have eight double-word entries. To handle the values between 1 and 8 that are not case-selection values, simply put the statement label of the `default` clause (or the label specifying the first instruction after the `endswitch` if there is no `default` clause) in each of the jump table entries that don't have a corresponding `case` clause. The following code demonstrates this technique:

```
readonly
     JmpTbl2: dword :=
                    [
                         &Case1, &Case2, &dfltCase, &Case4,
                         &dfltCase, &dfltCase, &dfltCase, &Case8
                    ];
          .
          .
          .
```

```
    cmp( ebx, 1 );
    jb dfltCase;
    cmp( ebx, 8 );
    ja dfltCase;
    jmp( JmpTbl2[ ebx*4 - 1*@size(dword) ] );

        Case1:
            stdout.put( "ebx = 1" );
            jmp EndOfSwitch;

        Case2:
            stdout.put( "ebx = 2" );
            jmp EndOfSwitch;

        Case4:
            stdout.put( "ebx = 4" );
            jmp EndOfSwitch;

        Case8:
            stdout.put( "ebx = 8" );
            jmp EndOfSwitch;

        dfltCase:
            stdout.put( "ebx is not 1, 2, 4, or 8" );

    EndOfSwitch:
```

There is a problem with this implementation of the switch statement. If the case values contain nonconsecutive entries that are widely spaced, the jump table could become exceedingly large. The following switch statement would generate an extremely large code file:

```
    switch( ebx )

        case( 1      ) << Stmt1 >>;
        case( 100    ) << Stmt2 >>;
        case( 1_000  ) << Stmt3 >>;
        case( 10_000 ) << Stmt4 >>;
        default << Stmt5 >>;

    endswitch;
```

In this situation, your program will be much smaller if you implement the switch statement with a sequence of if statements rather than using an indirect jump statement. However, keep one thing in mind—the size of the jump table does not normally affect the execution speed of the program. If the jump table contains two entries or two thousand, the switch statement will execute the multiway branch in a constant amount of time. The if statement implementation requires a linearly increasing amount of time for each case label appearing in the case statement.

Probably the biggest advantage to using assembly language over an HLL like Pascal or C/C++ is that you get to choose the actual implementation of statements like switch. In some instances you can implement a switch statement as a sequence of if..then..elseif statements, or you can implement it as a jump table, or you can use a hybrid of the two:

```
switch( eax )

     case( 0   ) << Stmt0 >>;
     case( 1   ) << Stmt1 >>;
     case( 2   ) << Stmt2 >>;
     case( 100 ) << Stmt3 >>;
     default << Stmt4 >>;

endswitch;
```

This could become

```
     cmp( eax, 100 );
     je DoStmt3;
     cmp( eax, 2 );
     ja TheDefaultCase;
     jmp( JmpTbl[ eax*4 ]);
     ...
```

Of course, HLA supports the following code high-level control structures:

```
 if( ebx = 100 ) then
     << Stmt3 >>;
else
     switch( eax )
          case(0) << Stmt0 >>;
          case(1) << Stmt1 >>;
          case(2) << Stmt2 >>;
          Otherwise << Stmt4 >>;
     endswitch;
endif;
```

But this tends to destroy the readability of the program. On the other hand, the extra code to test for 100 in the assembly language code doesn't adversely affect the readability of the program (perhaps because it's so hard to read already). Therefore, most people will add the extra code to make their program more efficient.

The C/C++ switch statement is very similar to the HLA switch statement. There is only one major semantic difference: The programmer must explicitly place a break statement in each case clause to transfer control to the first statement beyond the switch. This break corresponds to the jmp instruction at the end of each case sequence in the assembly code above. If the corresponding break is not present, C/C++ transfers control into the code of the following case. This is equivalent to leaving off the jmp at the end of the case's sequence:

```
     switch (i)
     {
          case 0: << Stmt1 >>;
          case 1: << Stmt2 >>;
          case 2: << Stmt3 >>;
               break;
          case 3: << Stmt4 >>;
               break;
          default: << Stmt5 >>;
     }
```

This translates into the following 80x86 code:

```
readonly
     JmpTbl: dword[4] := [ &case0, &case1, &case2, &case3 ];
          .
          .
          .
          mov( i, ebx );
          cmp( ebx, 3 );
          ja DefaultCase;
          jmp( JmpTbl[ ebx*4 ]);

               case0:
                    Stmt1;

               case1:
                    Stmt2;

               case2:
                    Stmt3;
                    jmp EndCase;    // Emitted for the break stmt.

               case3:
                    Stmt4;
                    jmp EndCase;    // Emitted for the break stmt.

               DefaultCase:
                    Stmt5;

          EndCase:
```

7.8 State Machines and Indirect Jumps

Another control structure commonly found in assembly language programs is the *state machine*. A state machine uses a *state variable* to control program flow. The FORTRAN programming language provides this capability with the assigned goto statement. Certain variants of C (for example, GNU's GCC from the Free Software Foundation) provide similar features. In assembly language, the indirect jump can implement state machines.

So what is a state machine? In very basic terms, it is a piece of code that keeps track of its execution history by entering and leaving certain "states." For the purposes of this chapter, we'll just assume that a state machine is a piece of code that (somehow) remembers the history of its execution (its *state*) and executes sections of code based on that history.

In a very real sense, all programs are state machines. The CPU registers and values in memory constitute the state of that machine. However, we'll use a much more constrained view. Indeed, for most purposes only a single variable (or the value in the EIP register) will denote the current state.

Now let's consider a concrete example. Suppose you have a procedure that you want to perform one operation the first time you call it, a different operation the second time you call it, yet something else the third time you call it, and then something new again on the fourth call. After the fourth call it repeats these four different operations in order. For example, suppose you want the procedure to add EAX and EBX the first time, subtract them on the second call, multiply them on the third, and divide them on the fourth. You could implement this procedure as follows:

```
procedure StateMachine;
static
     State:byte := 0;
begin StateMachine;

     cmp( State, 0 );
     jne TryState1;

          // State 0: Add ebx to eax and switch to State 1:

          add( ebx, eax );
          inc( State );
          exit StateMachine;

     TryState1:
     cmp( State, 1 );
     jne TryState2;

          // State 1: Subtract ebx from eax and switch to State 2:

          sub( ebx, eax );
          inc( State );       // State 1 becomes State 2.
          exit StateMachine;

     TryState2:
     cmp( State, 2 );
     jne MustBeState3;

          // If this is State 2, multiply ebx by eax and switch to State 3:

          intmul( ebx, eax );
          inc( State );       // State 2 becomes State 3.
          exit StateMachine;
     // If it isn't one of the above states, we must be in State 3,
     // so divide eax by ebx and switch back to State 0.

     MustBeState3:
     push( edx );         // Preserve this 'cause it gets whacked by div.
     xor( edx, edx );     // Zero extend eax into edx.
     div( ebx, edx:eax);
     pop( edx );          // Restore edx's value preserved above.
     mov( 0, State );     // Reset the state back to 0.

end StateMachine;
```

Technically, this procedure is not the state machine. Instead, it is the variable `State` and the `cmp/jne` instructions that constitute the state machine.

There is nothing particularly special about this code. It's little more than a `switch` statement implemented via the `if..then..elseif` construct. The only thing unique about this procedure is that it remembers how many times it has been called[4] and behaves differently depending upon the number of calls. While this is a *correct* implementation of the desired state machine, it is not particularly efficient. The astute reader, of course, would recognize that this code could be made a little faster using an actual `switch` statement rather than the `if..then..elseif` implementation. However, there is an even better solution.

A common implementation of a state machine in assembly language is to use an indirect jump. Rather than having a state variable that contains a value like 0, 1, 2, or 3, we could load the state variable with the *address* of the code to execute upon entry into the procedure. By simply jumping to that address, the state machine could save the tests needed to select the proper code fragment. Consider the following implementation using the indirect jump:

```
procedure StateMachine;
static
     State:dword := &State0;
begin StateMachine;

     jmp( State );

          // State 0: Add ebx to eax and switch to State 1:

     State0:
          add( ebx, eax );
          mov( &State1, State );
          exit StateMachine;

     State1:

          // State 1: Subtract ebx from eax and switch to State 2:
```

[4] Actually, it remembers how many times, `modulo 4`, that it has been called.

```
                sub( ebx, eax );
                mov( &State2, State );    // State 1 becomes State 2.
                exit StateMachine;

        State2:

                // If this is State 2, multiply ebx by eax and switch to State 3:

                intmul( ebx, eax );
                mov( &State3, State );    // State 2 becomes State 3.
                exit StateMachine;

        // State 3: Divide eax by ebx and switch back to State 0.

        State3:
                push( edx );         // Preserve this 'cause it gets whacked by div.
                xor( edx, edx );     // Zero extend eax into edx.
                div( ebx, edx:eax);
                pop( edx );            // Restore edx's value preserved above.
                mov( &State0, State ); // Reset the state back to 0.

end StateMachine;
```

The `jmp` instruction at the beginning of the `StateMachine` procedure transfers control to the location pointed at by the `State` variable. The first time you call `StateMachine` it points at the `State0` label. Thereafter, each subsection of code sets the `State` variable to point at the appropriate successor code.

7.9 Spaghetti Code

One major problem with assembly language is that it takes several statements to realize a simple idea encapsulated by a single high-level language statement. All too often an assembly language programmer will notice that she or he can save a few bytes or cycles by jumping into the middle of some program structure. After a few such observations (and corresponding modifications) the code contains a whole sequence of jumps in and out of portions of the code. If you were to draw a line from each jump to its destination, the resulting listing would end up looking like someone dumped a bowl of spaghetti on your code, hence the term *spaghetti code*.

Spaghetti code suffers from one major drawback—it's difficult (at best) to read such a program and figure out what it does. Most programs start out in a "structured" form only to become spaghetti code when sacrificed at the altar of efficiency. Alas, spaghetti code is rarely efficient. Because it's difficult to figure out exactly what's going on, it's very difficult to determine if you can use a better algorithm to improve the system. Hence, spaghetti code may wind up less efficient than structured code.

While it's true that producing some spaghetti code in your programs may improve its efficiency, doing so should always be a last resort after you've tried everything else and you still haven't achieved what you need. Always start out writing your programs with straightforward `if` and `switch` statements. Start combining sections of code (via `jmp` instructions) once everything is working and well understood. Of course, you should never obliterate the structure of your code unless the gains are worth it.

A famous saying in structured programming circles is, "After `goto`s, pointers are the next most dangerous element in a programming language." A similar saying is "Pointers are to data structures what `goto`s are to control structures." In other words, avoid excessive use of pointers. If pointers and `goto`s are bad, then the indirect jump must be the worst construct of all because it involves both `goto`s and pointers! Seriously, though, the indirect jump instruction should be avoided for casual use. Its use tends to make a program harder to read. After all, an indirect jump can (theoretically) transfer control to any point within a program. Imagine how hard it would be to follow the flow through a program if you have no idea what a pointer contains and you come across an indirect jump using that pointer. Therefore, you should always exercise care when using jump indirect instructions.

7.10 Loops

Loops represent the final basic control structure (sequences, decisions, and loops) that make up a typical program. Like so many other structures in assembly language, you'll find yourself using loops in places you've never dreamed of using loops. Most high-level languages have implied loop structures hidden away. For example, consider the BASIC statement `if A$ = B$ then 100`. This `if` statement compares two strings and jumps to statement 100 if they are equal. In assembly language, you would need to write a loop to compare each character in `A$` to the corresponding character in `B$` and then jump to statement 100 if and only if all the characters matched. In BASIC, there is no loop to be seen in the program. Assembly language requires a loop to compare the individual characters in the string.[5] This is but a small example that shows how loops seem to pop up everywhere.

Program loops consist of three components: an optional initialization component, an optional loop termination test, and the body of the loop. The order in which you assemble these components can dramatically affect the loop's operation. Three permutations of these components appear frequently in programs. Because of their frequency, these loop structures are given special names in high-level languages: `while` loops, `repeat..until` loops (`do..while` in C/C++), and infinite loops (e.g., `forever..endfor` in HLA).

[5] Of course, the HLA Standard Library provides the `str.eq` routine that compares the strings for you, effectively hiding the loop even in an assembly language program.

7.10.1 while Loops

The most general loop is the `while` loop. In HLA's high-level syntax it takes the following form:

```
while( expression ) do statements endwhile;
```

There are two important points to note about the `while` loop. First, the test for termination appears at the beginning of the loop. Second, as a direct consequence of the position of the termination test, the body of the loop may never execute if the boolean expression is always false.

Consider the following HLA `while` loop:

```
mov( 0, i );
while( i < 100 ) do

     inc( i );

endwhile;
```

The `mov( 0, i );` instruction is the initialization code for this loop. `i` is a loop-control variable, because it controls the execution of the body of the loop. `i < 100` is the loop termination condition. That is, the loop will not terminate as long as `i` is less than 100. The single instruction `inc( i );` is the loop body that executes on each loop iteration.

Note that an HLA `while` loop can be easily synthesized using `if` and `jmp` statements. For example, you may replace the previous HLA `while` loop with the following HLA code:

```
mov( 0, i );
WhileLp:
if( i < 100 ) then

     inc( i );
     jmp WhileLp;

endif;
```

More generally, you can construct any `while` loop as follows:

```
<< Optional initialization code >>

UniqueLabel:
if( not_termination_condition ) then

     << Loop body >>
     jmp UniqueLabel;

endif;
```

Therefore, you can use the techniques from earlier in this chapter to convert `if` statements to assembly language and add a single `jmp` instruction to produce a `while` loop. The example we've been looking at in this section translates to the following pure 80x86 assembly code:[6]

```
    mov( 0, i );
    WhileLp:
         cmp( i, 100 );
         jnl WhileDone;
         inc( i );
         jmp WhileLp;

    WhileDone:
```

7.10.2 repeat..until Loops

The `repeat..until` (`do..while`) loop tests for the termination condition at the end of the loop rather than at the beginning. In HLA high-level syntax, the `repeat..until` loop takes the following form:

```
    << Optional initialization code >>
    repeat

         << Loop body >>

    until( termination_condition );
```

This sequence executes the initialization code, then executes the loop body, and finally tests some condition to see if the loop should repeat. If the boolean expression evaluates to false, the loop repeats; otherwise the loop terminates. The two things you should note about the `repeat..until` loop are that the termination test appears at the end of the loop and, as a direct consequence of this, the loop body always executes at least once.

Like the `while` loop, the `repeat..until` loop can be synthesized with an `if` statement and a `jmp`. You could use the following:

```
    << Initialization code >>
    SomeUniqueLabel:

         << Loop body >>

    if( not_the_termination_condition ) then jmp SomeUniqueLabel; endif;
```

[6] Note that HLA will actually convert most `while` statements to different 80x86 code than this section presents. The reason for the difference appears in Section 7.11, when we explore how to write more efficient loop code.

Based on the material presented in the previous sections, you can easily synthesize `repeat..until` loops in assembly language. The following is a simple example:

```
     repeat

          stdout.put( "Enter a number greater than 100: " );
          stdin.get( i );

     until( i > 100 );

// This translates to the following if/jmp code:

     RepeatLabel:

          stdout.put( "Enter a number greater than 100: " );
          stdin.get( i );

     if( i <= 100 ) then jmp RepeatLabel; endif;

// It also translates into the following "pure" assembly code:

     RepeatLabel:

          stdout.put( "Enter a number greater than 100: " );
          stdin.get( i );

     cmp( i, 100 );
     jng RepeatLabel;
```

7.10.3 forever..endfor Loops

If `while` loops test for termination at the beginning of the loop and `repeat..until` loops check for termination at the end of the loop, the only place left to test for termination is in the middle of the loop. The HLA high-level `forever..endfor` loop, combined with the `break` and `breakif` statements, provides this capability. The `forever..endfor` loop takes the following form:

```
     forever

          << Loop body >>

     endfor;
```

Note that there is no explicit termination condition. Unless otherwise provided for, the `forever..endfor` construct forms an infinite loop. A `breakif` statement usually handles loop termination. Consider the following HLA code that employs a `forever..endfor` construct:

```
forever

     stdin.get( character );
     breakif( character = '.' );
     stdout.put( character );

endfor;
```

Converting a `forever` loop to pure assembly language is easy. All you need is a label and a `jmp` instruction. The `breakif` statement in this example is really nothing more than an `if` and a `jmp` instruction. The pure assembly language version of the code above looks something like the following:

```
foreverLabel:

     stdin.get( character );
     cmp( character, '.' );
     je ForIsDone;
     stdout.put( character );
     jmp foreverLabel;

ForIsDone:
```

7.10.4 for Loops

The `for` loop is a special form of the `while` loop that repeats the loop body a specific number of times. In HLA, the `for` loop takes the following form:

```
for( Initialization_Stmt; Termination_Expression; inc_Stmt ) do

     << statements >>

endfor;
```

This is completely equivalent to the following:

```
Initialization_Stmt;
while( Termination_Expression ) do

     << statements >>

     inc_Stmt;

endwhile;
```

Traditionally, programs use the for loop to process arrays and other objects accessed in sequential order. One normally initializes a loop-control variable with the initialization statement and then uses the loop-control variable as an index into the array (or other data type). For example:

```
for( mov( 0, esi ); esi < 7; inc( esi )) do

     stdout.put( "Array Element = ", SomeArray[ esi*4 ], nl );

endfor;
```

To convert this to pure assembly language, begin by translating the for loop into an equivalent while loop:

```
          mov( 0, esi );
          while( esi < 7 ) do

               stdout.put( "Array Element = ", SomeArray[ esi*4 ], nl );

               inc( esi );
          endwhile;
```

Now, using the techniques from the section on while loops, translate the code into pure assembly language:

```
          mov( 0, esi );
          WhileLp:
          cmp( esi, 7 );
          jnl EndWhileLp;

               stdout.put( "Array Element = ", SomeArray[ esi*4 ], nl );

               inc( esi );
               jmp WhileLp;

          EndWhileLp:
```

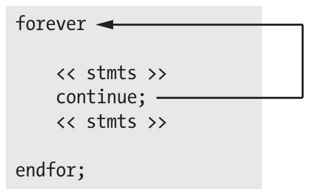
7.10.5 The break and continue Statements

The HLA break and continue statements both translate into a single jmp instruction. The break instruction exits the loop that immediately contains the break statement; the continue statement restarts the loop that immediately contains the continue statement.

Converting a break statement to pure assembly language is very easy. Just emit a jmp instruction that transfers control to the first statement following the end*xxxx* (or until) clause of the loop to exit. You can do this by placing a label after the associated end*xxxx* clause and jumping to that label. The following code fragments demonstrate this technique for the various loops.

```
// Breaking out of a FOREVER loop:

forever
     << stmts >>
          // break;
          jmp BreakFromForever;
     << stmts >>
endfor;
BreakFromForever:

// Breaking out of a FOR loop;
for( initStmt; expr; incStmt ) do
     << stmts >>
          // break;
          jmp BrkFromFor;
     << stmts >>
endfor;
BrkFromFor:

// Breaking out of a WHILE loop:

while( expr ) do
     << stmts >>
          // break;
          jmp BrkFromWhile;
     << stmts >>
endwhile;
BrkFromWhile:

// Breaking out of a REPEAT..UNTIL loop:

repeat
     << stmts >>
          // 20break;
          jmp BrkFromRpt;
     << stmts >>
until( expr );
BrkFromRpt:
```

The `continue` statement is slightly more complex than the `break` statement. The implementation is still a single `jmp` instruction; however, the target label doesn't wind up going in the same spot for each of the different loops. Figures 7-2, 7-3, 7-4, and 7-5 show where the `continue` statement transfers control for each of the HLA loops.



```
forever

    << stmts >>
    continue;
    << stmts >>

endfor;
```

Figure 7-2: `continue` destination for the `forever` loop

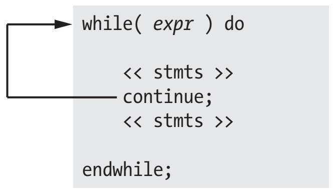


Figure 7-3: `continue` *destination and the* `while` *loop*

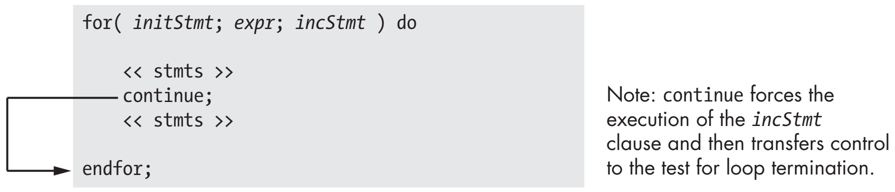

for(initStmt; expr; incStmt) do
<< stmts >>
continue;
<< stmts >>
endfor;
Note: continue forces the execution of the incStmt clause and then transfers control to the test for loop termination.

Figure 7-4: `continue` *destination and the* `for` *loop*

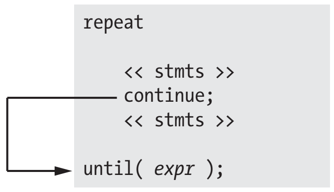


Figure 7-5: `continue` *destination and the* `repeat..until` *loop*

The following code fragments demonstrate how to convert the `continue` statement into an appropriate `jmp` instruction for each of these loop types.

forever..continue..endfor

```
// Conversion of forever loop with continue
// to pure assembly:
forever
     << stmts >>
     continue;
     << stmts >>
endfor;

// Converted code:

foreverLbl:
     << stmts >>
          // continue;
          jmp foreverLbl;
     << stmts >>
     jmp foreverLbl;
```

while..continue..endwhile

```
// Conversion of while loop with continue
// into pure assembly:
```

The Art of Assembly Language, 2nd Edition
(C) 2010 by Randall Hyde

```
while( expr ) do
     << stmts >>
     continue;
     << stmts >>
endwhile;

// Converted code:

whlLabel:
<< Code to evaluate expr >>
jcc EndOfWhile;         // Skip loop on expr failure.
     << stmts >>
          // continue;
          jmp whlLabel; // Jump to start of loop on continue.
     << stmts >>
     jmp whlLabel;      // Repeat the code.
EndOfwhile:
```

for..continue..endfor

```
// Conversion for a for loop with continue
// into pure assembly:

for( initStmt; expr; incStmt ) do
     << stmts >>
     continue;
     << stmts >>
endfor;

// Converted code:

initStmt
ForLpLbl:
<< Code to evaluate expr >>
jcc EndOfFor;           // Branch if expression fails.
     << stmts >>
          // continue;
          jmp ContFor;  // Branch to incStmt on continue.
     << stmts >>

     ContFor:
     incStmt
     jmp ForLpLbl;
EndOfFor:
```

repeat..continue..until

```
repeat
     << stmts >>
     continue;
     << stmts >>
until( expr );
```

```
// Converted code:

RptLpLbl:
     << stmts >>
          // continue;
          jmp ContRpt;  // Continue branches to loop termination test.
          << stmts >>
     ContRpt:
     << Code to test expr >>
     jcc RptLpLbl;      // Jumps if expression evaluates false.
```

7.10.6 Register Usage and Loops

Given that the 80x86 accesses registers more efficiently than memory locations, registers are the ideal spot to place loop-control variables (especially for small loops). However, there are some problems associated with using registers within a loop. The primary problem with using registers as loop-control variables is that registers are a limited resource. The following will not work properly because it attempts to reuse a register (CX) that is already in use:

```
          mov( 8, cx );
          loop1:
               mov( 4, cx );
               loop2:
                    << stmts >>
                    dec( cx );
                    jnz loop2;
               dec( cx );
           jnz loop1;
```

The intent here, of course, was to create a set of nested loops, that is, one loop inside another. The inner loop (`loop2`) should repeat four times for each of the eight executions of the outer loop (`loop1`). Unfortunately, both loops use the same register as a loop-control variable. Therefore, this will form an infinite loop because CX will contain 0 at the end of the first loop. Because CX is always 0 upon encountering the second `dec` instruction, control will always transfer to the `loop1` label (because decrementing 0 produces a nonzero result). The solution here is to save and restore the CX register or to use a different register in place of CX for the outer loop:

```
          mov( 8, cx );
          loop1:
               push( cx );
               mov( 4, cx );
               loop2:
                    << stmts >>
                    dec( cx );
                    jnz loop2;

               pop( cx );
```

```
            dec( cx );
            jnz loop1;
```

or

```
        mov( 8, dx );
        loop1:
            mov( 4, cx );
            loop2:
                << stmts >>
                dec( cx );
                jnz loop2;

            dec( dx );
            jnz loop1;
```

Register corruption is one of the primary sources of bugs in loops in assembly language programs, so always keep an eye out for this problem.

7.11 Performance Improvements

The 80x86 microprocessors execute sequences of instructions at blinding speed. Therefore, you'll rarely encounter a slow program that doesn't contain any loops. Because loops are the primary source of performance problems within a program, they are the place to look when attempting to speed up your software. While a treatise on how to write efficient programs is beyond the scope of this chapter, there are some things you should be aware of when designing loops in your programs. They're all aimed at removing unnecessary instructions from your loops in order to reduce the time it takes to execute a single iteration of the loop.

7.11.1 Moving the Termination Condition to the End of a Loop

Consider the following flow graphs for the three types of loops presented earlier:

```
repeat..until loop:
    Initialization code
        Loop body
    Test for termination
    Code following the loop

while loop:
    Initialization code
    Loop termination test
        Loop body
        Jump back to test
    Code following the loop
```

```
forever..endfor loop:
     Initialization code
          Loop body part one
          Loop termination test
          Loop body part two
          Jump back to Loop body part one
     Code following the loop
```

As you can see, the `repeat..until` loop is the simplest of the bunch. This is reflected in the assembly language implementation of these loops. Consider the following `repeat..until` and `while` loops that are semantically identical:

```
// Example involving a WHILE loop:

     mov( edi, esi );
     sub( 20, esi );
     while( esi <= edi ) do

          << stmts >>
          inc( esi );

     endwhile;

// Conversion of the code above into pure assembly language:

     mov( edi, esi );
     sub( 20, esi );
     whlLbl:
     cmp( esi, edi );
     jnle EndOfWhile;

          << stmts >>
          inc( esi );
          << stmts >>
          jmp whlLbl;

     EndOfWhile:


// Example involving a REPEAT..UNTIL loop:

     mov( edi, esi );
     sub( 20, esi );
     repeat

          << stmts >>
          inc( esi );

     until( esi > edi );
```

```
// Conversion of the REPEAT..UNTIL loop into pure assembly:

     rptLabel:
          << stmts >>
          inc( esi );
          cmp( esi, edi );
          jng rptLabel;
```

As you can see by carefully studying the conversion to pure assembly language, testing for the termination condition at the end of the loop allowed us to remove a `jmp` instruction from the loop. This can be significant if this loop is nested inside other loops. In the preceding example there wasn't a problem with executing the body at least once. Given the definition of the loop, you can easily see that the loop will be executed exactly 20 times. This suggests that the conversion to a `repeat..until` loop is trivial and always possible. Unfortunately, it's not always quite this easy. Consider the following HLA code:

```
     while( esi <= edi ) do
          << stmts >>
          inc( esi );
     endwhile;
```

In this particular example, we haven't the slightest idea what ESI contains upon entry into the loop. Therefore, we cannot assume that the loop body will execute at least once. So we must test for loop termination before executing the body of the loop. The test can be placed at the end of the loop with the inclusion of a single `jmp` instruction:

```
     jmp WhlTest;
     TopOfLoop:
          << stmts >>
          inc( esi );
     WhlTest:
          cmp( esi, edi );
          jle TopOfLoop;
```

Although the code is as long as the original `while` loop, the `jmp` instruction executes only once rather than on each repetition of the loop. Note that this slight gain in efficiency is obtained via a slight loss in readability. The second code sequence above is closer to spaghetti code than the original implementation. Such is often the price of a small performance gain. Therefore, you should carefully analyze your code to ensure that the performance boost is worth the loss of clarity. More often than not, assembly language programmers sacrifice clarity for dubious gains in performance, producing impossible-to-understand programs.

Note, by the way, that HLA translates its high-level `while` statement into a sequence of instructions that test the loop termination condition at the bottom of the loop using exactly the technique this section describes.

7.11.2 Executing the Loop Backwards

Because of the nature of the flags on the 80x86, loops that repeat from some number down to (or up to) 0 are more efficient than loops that execute from 0 to some other value. Compare the following HLA `for` loop and the code it generates:

```
for( mov( 1, j ); j <= 8; inc( j ) ) do
     << stmts >>
endfor;

// Conversion to pure assembly (as well as using a REPEAT..UNTIL form):

mov( 1, j );
ForLp:
     << stmts >>
     inc( j );
     cmp( j, 8 );
     jnge ForLp;
```

Now consider another loop that also has eight iterations but runs its loop-control variable from 8 down to 1 rather than 1 up to 8:

```
mov( 8, j );
LoopLbl:
     << stmts >>
     dec( j );
     jnz LoopLbl;
```

Note that by running the loop from 8 down to 1 we saved a comparison on each repetition of the loop.

Unfortunately, you cannot force all loops to run backward. However, with a little effort and some coercion you should be able to write many `for` loops so that they operate backward. Saving the execution time of the `cmp` instruction on each iteration of the loop may result in faster code.

The example above worked out well because the loop ran from 8 down to 1. The loop terminated when the loop-control variable became 0. What happens if you need to execute the loop when the loop-control variable goes to 0? For example, suppose that the loop above needed to range from 7 down to 0. As long as the upper bound is positive, you can substitute the `jns` instruction in place of the `jnz` instruction in the earlier code:

```
mov( 7, j );
LoopLbl:
     << stmts >>
     dec( j );
     jns LoopLbl;
```

This loop will repeat eight times, with `j` taking on the values 7..0. When it decrements 0 to −1, it sets the sign flag and the loop terminates.

Keep in mind that some values may look positive but are actually negative. If the loop-control variable is a byte, then values in the range 128..255 are negative in the two's complement system. Therefore, initializing the loop-control variable with any 8-bit value in the range 129..255 (or, of course, 0) terminates the loop after a single execution. This can get you into trouble if you're not careful.

7.11.3 Loop-Invariant Computations

A *loop-invariant computation* is some calculation that appears within a loop that always yields the same result. You needn't do such computations inside the loop. You can compute them outside the loop and reference the value of the computations inside the loop. The following HLA code demonstrates an invariant computation:

```
     for( mov( 0, eax ); eax < n; inc( eax )) do

          mov( eax, edx );
          add( j, edx );
          sub( 2, edx );
          add( edx, k );

     endfor;
```

Because j never changes throughout the execution of this loop, the subexpression j-2 can be computed outside the loop:

```
     mov( j, ecx );
     sub( 2, ecx );
     for( mov( 0, eax ); eax < n; inc( eax )) do

          mov( eax, edx );
          add( ecx, edx );
          add( edx, k );

     endfor;
```

Although we've eliminated a single instruction by computing the subexpression j-2 outside the loop, there is still an invariant component to this calculation. Note that this invariant component executes *n* times in the loop; this means that we can translate the previous code to the following:

```
     mov( j, ecx );
     sub( 2, ecx );
     intmul( n, ecx );   // Compute n*(j-2) and add this into k outside
     add( ecx, k );      // the loop.
     for( mov( 0, eax ); eax < n; inc( eax )) do
```

```
        add( eax, k );

    endfor;
```

As you can see, we've shrunk the loop body from four instructions down to one. Of course, if you're really interested in improving the efficiency of this particular loop, you can compute the result without using a loop at all (there is a formula that corresponds to the iterative calculation above). Still, this simple example demonstrates elimination of loop-invariant calculations from a loop.

7.11.4 Unraveling Loops

For small loops, that is, those whose body is only a few statements, the overhead required to process a loop may constitute a significant percentage of the total processing time. For example, look at the following Pascal code and its associated 80x86 assembly language code:

```
    for i := 3 downto 0 do A[i] := 0;

    mov( 3, i );
    LoopLbl:
        mov( i, ebx );
        mov( 0, A[ ebx*4 ] );
        dec( i );
        jns LoopLbl;
```

Four instructions execute on each repetition of the loop. Only one instruction is doing the desired operation (moving a 0 into an element of A). The remaining three instructions control the loop. Therefore, it takes 16 instructions to do the operation logically required by 4.

While there are many improvements we could make to this loop based on the information presented thus far, consider carefully exactly what it is that this loop is doing—it's storing four 0s into A[0] through A[3]. A more efficient approach is to use four mov instructions to accomplish the same task. For example, if A is an array of double words, then the following code initializes A much faster than the code above:

```
    mov( 0, A[0] );
    mov( 0, A[4] );
    mov( 0, A[8] );
    mov( 0, A[12] );
```

Although this is a simple example, it shows the benefit of *loop unraveling* (also known as *loop unrolling*). If this simple loop appeared buried inside a set of nested loops, the 4:1 instruction reduction could possibly double the performance of that section of your program.

Of course, you cannot unravel all loops. Loops that execute a variable number of times are difficult to unravel because there is rarely a way to determine (at assembly time) the number of loop iterations. Therefore, unraveling a loop is a process best applied to loops that execute a known number of times (and the number of times is known at assembly time).

Even if you repeat a loop some fixed number of iterations, it may not be a good candidate for loop unraveling. Loop unraveling produces impressive performance improvements when the number of instructions controlling the loop (and handling other overhead operations) represents a significant percentage of the total number of instructions in the loop. Had the previous loop contained 36 instructions in the body (exclusive of the 4 overhead instructions), then the performance improvement would be, at best, only 10 percent (compared with the 300–400 percent it now enjoys). Therefore, the costs of unraveling a loop, that is, all the extra code that must be inserted into your program, quickly reach a point of diminishing returns as the body of the loop grows larger or as the number of iterations increases. Furthermore, entering that code into your program can become quite a chore. Therefore, loop unraveling is a technique best applied to small loops.

Note that the superscalar 80x86 chips (Pentium and later) have *branch-prediction hardware* and use other techniques to improve performance. Loop unrolling on such systems may actually *slow down* the code because these processors are optimized to execute short loops.

7.11.5 Induction Variables

Consider the following loop:

```
    for i := 0 to 255 do csetVar[i] := {};
```

Here the program is initializing each element of an array of character sets to the empty set. The straightforward code to achieve this is the following:

```
mov( 0, i );
FLp:

    // Compute the index into the array (note that each element
    // of a CSET array contains 16 bytes).

    mov( i, ebx );
    shl( 4, ebx );

    // Set this element to the empty set (all 0 bits).

    mov( 0, csetVar[ ebx ] );
    mov( 0, csetVar[ ebx+4 ] );
    mov( 0, csetVar[ ebx+8 ] );
    mov( 0, csetVar[ ebx+12 ] );
```

```
     inc( i );
     cmp( i, 256 );
     jb FLp;
```

Although unraveling this code will still produce a performance improvement, it will take 1,024 instructions to accomplish this task, too many for all but the most time-critical applications. However, you can reduce the execution time of the body of the loop using *induction variables*. An induction variable is one whose value depends entirely on the value of some other variable. In the example above, the index into the array `csetVar` tracks the loop-control variable (it's always equal to the value of the loop-control variable times 16). Because `i` doesn't appear anywhere else in the loop, there is no sense in performing the computations on `i`. Why not operate directly on the array index value? The following code demonstrates this technique:

```
mov( 0, ebx );
FLp:
     mov( 0, csetVar[ ebx ]);
     mov( 0, csetVar[ ebx+4 ] );
     mov( 0, csetVar[ ebx+8 ] );
     mov( 0, csetVar[ ebx+12 ] );

     add( 16, ebx );
     cmp( ebx, 256*16 );
     jb FLp;
```

The induction that takes place in this example occurs when the code increments the loop-control variable (moved into EBX for efficiency reasons) by 16 on each iteration of the loop rather than by 1. Multiplying the loop-control variable by 16 (and also the final loop-termination constant value) allows the code to eliminate multiplying the loop-control variable by 16 on each iteration of the loop (that is, this allows us to remove the `shl` instruction from the previous code). Further, because this code no longer refers to the original loop-control variable (`i`), the code can maintain the loop-control variable strictly in the EBX register.

7.12 Hybrid Control Structures in HLA

The HLA high-level language control structures have a few drawbacks: (1) they're not true assembly language instructions, (2) complex boolean expressions support only short-circuit evaluation, and (3) they often introduce inefficient coding practices into a language that most people use only when they need to write high-performance code. On the other hand, while the 80x86 low-level control structures let you write efficient code, the resulting code is very difficult to read and maintain. HLA provides a set of hybrid control structures that allow you to use pure assembly language statements to evaluate boolean expressions while using the high-level control structures to delineate the statements controlled by the boolean expressions. The result is code that is much more readable than pure assembly language without being a whole lot less efficient.

HLA provides hybrid forms of the `if..elseif..else..endif`, `while..endwhile`, `repeat..until`, `breakif`, `exitif`, and `continueif` statements (that is, those that involve a boolean expression). For example, a hybrid `if` statement takes the following form:

```
     if( #{ instructions }# ) then statements endif;
```

Note the use of `#{` and `}#` operators to surround a sequence of instructions within this statement. This is what differentiates the hybrid control structures from the standard high-level language control structures. The remaining hybrid control structures take the following forms:

```
while( #{ statements }# ) statements endwhile;
repeat statements until( #{ statements }# );
breakif( #{ statements }# );
exitif( #{ statements }# );
continueif( #{ statements }# );
```

The statements within the curly braces replace the normal boolean expression in an HLA high-level control structure. These particular statements are special insofar as HLA defines two pseudo-labels, `true` and `false`, within their context. HLA associates the label `true` with the code that would normally execute if a boolean expression were present and that expression's result was true. Similarly, HLA associates the label `false` with the code that would execute if a boolean expression in one of these statements evaluated false. As a simple example, consider the following two (equivalent) `if` statements:

```
if( eax < ebx ) then inc( eax ); endif;

if
( #{
     cmp( eax, ebx );
     jnb false;
}# ) then
     inc( eax );

endif;
```

The `jnb` that transfers control to the `false` label in this latter example will skip over the `inc` instruction if EAX is not less than EBX. Note that if EAX is less than EBX, then control falls through to the `inc` instruction. This is roughly equivalent to the following pure assembly code:

```
cmp( eax, ebx );
jnb falseLabel;
     inc( eax );
falseLabel:
```

As a slightly more complex example, consider the statement

```
if( eax >= j && eax <= k ) then sub( j, eax ); endif;
```

The following hybrid `if` statement accomplishes the above:

```
if
( #{
     cmp( eax, j );
     jnae false;
     cmp( eax, k );
     jnae false;
}# ) then
     sub( j, eax );

endif;
```

As one final example of the hybrid `if` statement, consider the following:

```
// if( ((eax > ebx) && (eax < ecx)) || (eax = edx)) then
//     mov( ebx, eax );
// endif;

if
( #{
     cmp( eax, edx );
     je true;
     cmp( eax, ebx );
     jng false;
     cmp( eax, ecx );
     jnb false;
}# ) then
     mov( ebx, eax );

endif;
```

Because these examples are rather trivial, they don't really demonstrate how much more readable the code can be when using hybrid statements rather than pure assembly code. However, one thing you should notice is that using hybrid statements eliminates the need to insert labels throughout your code. This can make your programs easier to read and understand.

For the `if` statement, the `true` label corresponds to the `then` clause of the statement; the `false` label corresponds to the `elseif`, `else`, or `endif` clause (whichever follows the `then` clause). For the `while` loop, the `true` label corresponds to the body of the loop, whereas the `false` label is attached to the first statement following the corresponding `endwhile`. For the `repeat..until` statement, the `true` label is attached to the code following the `until` clause, whereas the `false` label is attached to the first statement of the body of the loop. The `breakif`, `exitif`, and `continueif` statements associate the `false` label with the statement immediately following one of these statements; they associate the `true` label with the code normally associated with a `break`, `exit`, or `continue` statement.

7.13 For More Information

HLA contains a few additional high-level control structures beyond those this chapter describes. Examples include the `try..endtry` block and the `foreach` statement. A discussion of these statements does not appear in this chapter because these are advanced control structures and their implementation is too complex to describe this early in the text. For more information on their implementation, see the electronic edition at *http://www.artofasm.com/* (or *http://webster.cs.ucr.edu/*) or the HLA reference manual.