HLA Code Generation for HLL Statements Appendix L

One of the principal advantages of using assembly language over high level languages is the control that assembly provides. High level languages (HLLs) represent an abstraction of the underlying hardware. Those who write HLL code give up this control in exchange for the engineering efficiencies enjoyed by HLL programmers. Some advanced HLL programmers (who have a good mastery of the underlying machine architecture) are capable of writing fairly efficient programs by recognizing what the compiler does with various high level control constructs and choosing the appropriate construct to emit the machine code they want. While this "low-level programming in a high level language" does leave the programmer at the mercy of the compiler-writer, it does provide a mechanism whereby HLL programmers can write more efficient code by chosing those HLL constructs that compile into efficient machine code.

Although the High Level Assembler (HLA) allows a programmer to work at a very low level, HLA also provides structured high-level control constructs that let assembly programmers use higher-level code to help make their assembly code more readable. Those assembly language programmers who need (or want) to exercise maximum control over their programs will probably want to avoid using these statements since they tend to obscure what is happening at a really low level. At the other extreme, those who would always use these high-level control structures might question if they really want to use assembly language in their applications; after all, if they're writing high level code, perhaps they should use a high level language and take advantage of optimizing technology and other fancy features found in modern compilers. Between these two extremes lies the typical assembly language programmer. The one who realizes that most code doesn't need to be super-efficient and is more interested in productively producing lots of software rather than worrying about how many CPU cycles the one-time initialization code is going to consume. HLA is perfect for this type of programmer because it lets you work at a high level of abstraction when writing code whose performance isn't an issue and it lets you work at a low level of abstraction when working on code that requires special attention.

Between code whose performance doesn't matter and code whose performance is critical lies a big gray region: code that should be reasonably fast but speed isn't the number one priority. Such code needs to be reasonably readable, maintainable, and as free of defects as possible. In other words, code that is a good candidate for using high level control and data structures if their use is reasonably efficient.

Unlike various HLL compilers, HLA does not (yet!) attempt to optimize the code that you write. This puts HLA at a disadvantage: it relies on the optimizer between your ears rather than the one supplied with the compiler. If you write sloppy high level code in HLA then a HLL version of the same program will probably be more efficient if it is compiled with a decent HLL compiler. For code where performance matters, this can be a disturbing revelation (you took the time and bother to write the code in assembly but an equivalent C/C++ program is faster). The purpose of this appendix is to describe HLA's code generation in detail so you can intelligently choose when to use HLA's high level features and when you should stick with low-level assembly language.

L.1 The HLA Standard Library

The HLA Standard Library was designed to make learning assembly language programming easy for beginning programmers. Although the code in the library isn't terrible, very little effort was made to write top-performing code in the library. At some point in the future this may change as work on the library progresses, but if you're looking to write very high-performance code you should probably avoid calling routines in the HLA Standard Library from (speed) critical sections of your program.

Don't get the impression from the previous paragraph that HLA's Standard Library contains a bunch of slow-poke routines, however. Many of the HLA Standard Library routines use decent algorithms and data structures so they perform quite well in typical situations. For example, the HLA string format is far more efficient than strings in C/C++. The world's best C/C++ *strlen* routine is almost always going to be slower than HLA *str.len* function. This is because HLA uses a better definition for string data than C/C++, it has little to do with the actual implementation of the *str.len* code. This is not to say that HLA's *str.len* routine cannot be improved; but the routine is very fast already.

One problem with using the HLA Standard Library is the frame of mind it fosters during the development of a program. The HLA Standard Library is strongly influenced by the C/C++ Standard Library and libraries common in other high level languages. While the HLA Standard Library is a wonderful tool that can help you write assembly code faster than ever before, it also encourages you to think at a higher level. As any expert assembly language programmer can tell you, the real benefits of using assembly language occur only when you "think in assembly" rather than in a high level language. No matter how efficient the routines in the Standard Library happen to be, if you're "writing C++ programs with MOV instructions" the result is going to be little better than writing the code in C++ to begin with.

One unfortunate aspect of the HLA Standard Library is that it encourages you to think at a higher level and you'll often miss a far more efficient low-level solution as a result. A good example is the set of string routines in the HLA Standard Library. If you use those routines, even if they were written as efficiently as possible, you may not be writing the fastest possible program you can because you've limited your thinking to string objects which are a higher level abstraction. If you did not have the HLA Standard Library laying around and you had to do all the character string manipulation yourself, you might choose to treat the objects as character arrays in memory. This change of perspective can produce dramatic performance improvement under certain circumstances.

The bottom line is this: the HLA Standard Library is a wonderful collection of routines and they're not particularly inefficient. They're very easy and convenient to use. However, don't let the HLA Standard Library lull you into choosing data structures or algorithms that are not the most appropriate for a given section of your program.

L.2 Compiling to MASM Code -- The Final Word

The remainder of this document will discuss, in general, how HLA translates various HLL-style statements into assembly code. Sometimes a general discussion may not provide specific answers you need about HLA's code generation capabilities. Should you have a specific question about how HLA generates code with respect to a given code sequence, you can always run the compiler and observe the output it produces. To do this, it is best to create a simple program that contains only the construct you wish to study and compile that program to assembly code. For example, consider the following very simple HLA program:

```
program t;

begin t;

    if( eax = 0 ) then

        mov( 1, eax );

    endif;

end t;
```

If you compile this program using the command window prompt "hla -s t.hla" then HLA produces a (MASM) file similar to the following[1]:

```
        if      @Version lt 612
        .586
        else
        .686
        .mmx
        .xmm
```

1. Because the code generator in HLA is changing all the time, this file may not reflect an accurate compilation of the above HLA code. However, the concepts will be the same.


```
                endif
                .model  flat, syscall
offset32        equ     <offset flat:>
                assume  fs:nothing
?ExceptionPtr   equ     <(dword ptr fs:[0])>
                externdef ??HWexcept:near32
                externdef ??Raise:near32

std_output_hndl equ     -11

                externdef __imp__ExitProcess@4:dword
                externdef __imp__GetStdHandle@4:dword
                externdef __imp__WriteFile@20:dword

cseg            segment page public 'code'
cseg            ends
readonly        segment page public 'data'
readonly        ends
strings         segment page public 'data'
strings         ends
dseg            segment page public 'data'
dseg            ends
bssseg          segment page public 'data'
bssseg          ends


strings         segment page public 'data'

?dfltmsg        byte    "Unhandled exception error.",13,10
?dfltmsgsize    equ     34
?absmsg        byte    "Attempted call of abstract procedure or method.",13,10
?absmsgsize     equ     55
strings         ends
dseg            segment page public 'data'
?dfmwritten     word    0
?dfmStdOut      dword   0

                public  ?MainPgmCoroutine
?MainPgmCoroutine byte 0 dup (?)
                dword   ?MainPgmVMT
                dword   0       ;CurrentSP
                dword   0       ;Pointer to stack
                dword   0       ;ExceptionContext
                dword   0       ;Pointer to last caller
?MainPgmVMT     dword   ?QuitMain
dseg            ends
cseg            segment page public 'code'

?QuitMain       proc    near32
                pushd   1
                call    dword ptr __imp__ExitProcess@4

?QuitMain       endp

cseg            ends
```



```
cseg            segment page public 'code'

??DfltExHndlr   proc    near32

                pushd   std_output_hndl
                call    __imp__GetStdHandle@4
                mov     ?dfmStdOut, eax
                pushd   0       ;lpOverlapped
                pushd   offset32 ?dfmwritten    ;BytesWritten
                pushd   ?dfltmsgsize    ;nNumberOfBytesToWrite
                pushd   offset32 ?dfltmsg       ;lpBuffer
                pushd   ?dfmStdOut              ;hFile
                call    __imp__WriteFile@20

                pushd   0
                call    dword ptr __imp__ExitProcess@4

??DfltExHndlr   endp

                public  ??Raise
??Raise proc    near32
                jmp     ??DfltExHndlr
??Raise endp

                public  ??HWexcept
??HWexcept      proc    near32
                mov     eax, 1
                ret
??HWexcept      endp

?abstract       proc    near32

                pushd   std_output_hndl
                call    __imp__GetStdHandle@4
                mov     ?dfmStdOut, eax
                pushd   0       ;lpOverlapped
                pushd   offset32 ?dfmwritten    ;BytesWritten
                pushd   ?absmsgsize     ;nNumberOfBytesToWrite
                pushd   offset32 ?absmsg        ;lpBuffer
                pushd   ?dfmStdOut              ;hFile
                call    __imp__WriteFile@20

                pushd   0
                call    dword ptr __imp__ExitProcess@4

?abstract       endp


                public  ?HLAMain
?HLAMain        proc    near32


; Set up the Structured Exception Handler record
; for this program.

                push    offset32 ??DfltExHndlr
                push    ebp
```



```
                push    offset32 ?MainPgmCoroutine
                push    offset32 ??HWexcept
                push    ?ExceptionPtr
                mov     ?ExceptionPtr, esp
                mov     dword ptr ?MainPgmCoroutine+12, esp

                pushd   0               ;No Dynamic Link.
                mov     ebp, esp        ;Pointer to Main's locals
                push    ebp             ;Main's display.
                mov     [ebp+16], esp
                cmp     eax, 0
                jne     ?1_false
                mov     eax, 1
?1_false:
                push    0
                call    dword ptr __imp__ExitProcess@4
?HLAMain        endp
cseg            ends
                end
```

The code of interest in this example is at the very end, after the comment ";Main's display" appears in the text. The actual code sequence that corresponds to the IF statement in the main program is the following:

```
                   cmp     eax, 0
                jne     ?1_false
                mov     eax, 1
?1_false:
```

Note: you can verify that this is the code emitted by the IF statement by simply removing the IF, recompiling, and comparing the two assembly outputs. You'll find that the only difference between the two assembly output files is the four lines above. Another way to "prove" that this is the code sequence emitted by the HLA IF statement is to insert some comments into the assembly output file using HLA's #ASM..#ENDASM directives. Consider the following modification to the "t.hla" source file:

```
program t;

begin t;

    #asm
    ; Start of IF statement:
    #endasm

    if( eax = 0 ) then

        mov( 1, eax );

    endif;

    #asm
    ; End if IF statement.
    #endasm
```



```
end t;
```

HLA's #asm directive tells the compiler to simply emit everything between the #asm and #endasm keywords directly to the assembly output file. In this example the HLA program uses these directives to emit a pair of comments that will bracket the code of interest in the output file. Compiling this to assembly code (and stripping out the irrelevant stuff before the HLA main program) yields the following:

```
                public  ?HLAMain
?HLAMain        proc    near32


; Set up the Structured Exception Handler record
; for this program.

                push    offset32 ??DfltExHndlr
                push    ebp
                push    offset32 ?MainPgmCoroutine
                push    offset32 ??HWexcept
                push    ?ExceptionPtr
                mov     ?ExceptionPtr, esp
                mov     dword ptr ?MainPgmCoroutine+12, esp

                pushd   0               ;No Dynamic Link.
                mov     ebp, esp        ;Pointer to Main's locals
                push    ebp             ;Main's display.
                mov     [ebp+16], esp

;#asm


        ; Start of IF statement:
        ;#endasm

                cmp     eax, 0
                jne     ?1_false
                mov     eax, 1
?1_false:

;#asm


        ; End if IF statement.
        ;#endasm

                push    0
                call    dword ptr __imp__ExitProcess@4
?HLAMain        endp
cseg            ends
                end
```

This technique (embedding bracketing comments into the assembly output file) is very useful if it is not possible to isolate a specific statement in its own source file when you want to see what HLA does during compilation.

L.3 The HLA if..then..endif Statement, Part I

Although the HLA IF statement is actually one of the more complex statements the compiler has to deal with (in terms of how it generates code), the IF statement is probably the first statement that comes to mind when something thinks about high level control structures. Furthermore, you can implement most of the other control structures if you have an IF and a GOTO (JMP) statement, so it makes sense to discuss the IF statement first. Nevertheless, there is a bit of complexity that is unnecessary at this point, so we'll begin our discussion with a simplified version of the IF statement; for this simplified version we'll not consider the ELSEIF and ELSE clauses of the IF statement.

The basic HLA IF statement uses the following syntax:

```
if( simple_boolean_expression ) then

    << statements to execute if the expression evaluates true >>

endif;
```

At the machine language level, what the compiler needs to generate is code that does the following:

```
<< Evaluate the boolean expression >>

<< Jump around the following statements if the expression was false >>

<< statements to execute if the expression evaluates true >>

<< Jump to this point if the expression was false >>
```

The example in the previous section is a good demonstration of what HLA does with a simple IF statement. As a reminder, the HLA program contained

```
    if( eax = 0 ) then

        mov( 1, eax );

    endif;
```

and the HLA compiler generated the following assembly language code:

```
                cmp     eax, 0
                jne     ?1_false
                mov     eax, 1
?1_false:
```

Evaluation of the boolean expression was accomplished with the single "cmp eax, 0" instruction. The "jne ?1_false" instruction jumps around the "mov eax, 1" instruction (which is the statement to execute if the expression evaluates true) if the expression evaluates false. Conversely, if EAX is equal to zero, then the code falls through to the MOV instruction. Hence the semantics are exactly what we want for this high level control structure.

HLA automatically generates a unique label to branch to for each IF statement. It does this properly even if you nest IF statements. Consider the following code:

```
program t;

begin t;

    if( eax > 0 ) then

        if( eax < 10 ) then

            inc( eax );

        endif;

    endif;


end t;
```

The code above generates the following assembly output:

```
                cmp     eax, 0
                jna     ?1_false
                cmp     eax, 10
                jnb     ?2_false
                inc     eax
?2_false:
?1_false:
```

As you can tell by studying this code, the INC instruction only executes if the value in EAX is greater than zero and less than ten.

Thus far, you can see that HLA's code generation isn't too bad. The code it generates for the two examples above is roughly what a good assembly language programmer would write for approximately the same semantics.

L.4 Boolean Expressions in HLA Control Structures

The HLA IF statement and, indeed, most of the HLA control structures rely upon the evaluation of a boolean expression in order to direct the flow of the program. Unlike high level languages, HLA restricts boolean expressions in control structures to some very simple forms. This was done for two reasons: (1) HLA's design frowns upon side effects like register modification in the compiled code, and (2) HLA is intended for use by beginning assembly language students; the restricted boolean expression model is closer to the low level machine architecture and it forces them to start thinking in these terms right away.

With just a few exceptions, HLA's boolean expressions are limited to what HLA can easily compile to a CMP and a condition jump instruction pair or some other simple instruction sequence. Specifically, HLA allows the following boolean expressions:

operand1 relop operand2

relop is one of:

```
=  or ==        (either one, both are equivalent)
<> or !=        (either one, both are equivalent)
<
<=
>
>=
```

In the expressions above $operand_1$ and $operand_2$ are restricted to those operands that are legal in a CMP instruction. This is because HLA translates expressions of this form to the two instruction sequence:

```
cmp( operand1, operand2 );
jXX someLabel;
```

where "jXX" represents some condition jump whose sense is the opposite of that of the expression (e.g., "eax > ebx" generates a "JNA" instruction since "NA" is the opposite of ">").

Assuming you want to compare the two operands and jump around some sequence of instructions if the relationship does not hold, HLA will generate fairly efficient code for this type of expression. One thing you should watch out for, though, is that HLA's high level statements (e.g., IF) make it very easy to write code like the following:

```
if( i = 0 ) then

   ...

elseif( i = 1 ) then

    ...

elseif( i = 2 ) then

    ...
.
.
.
endif;
```

This code looks fairly innocuous, but the programmer who is aware of the fact that HLA emits the following would probably not use the code above:

```
     cmp( i, 0 );
     jne lbl;
       .
       .
       .
lbl: cmp( i, 1 );
     jne lbl2;
       .
       .
       .
lbl2: cmp( i, 2 );
```



```
.
.
.
```

A good assembly language programmer would realize that it's much better to load the variable "i" into a register and compare the register in the chain of CMP instructions rather than compare the variable each time. The high level syntax slightly obscures this problem; just one thing to be aware of.

HLA's boolean expressions do not support conjunction (logical AND) and disjunction (logical OR). The HLA programmer must manually synthesize expressions involving these operators. Doing so forces the programmer to link in lower level terms, which is usually more efficient. However, there are many common expressions involving conjunction that HLA could efficiently compile into assembly language. Perhaps the most common example is a test to see if an operand is within (or outside) a range specified by two constants. In a HLL like C/C++ you would typically use an expression like "(value >= low_constant && value <= high_constant)" to test this condition. HLA allows four special boolean expressions that check to see if a register or a memory location is within a specified range. The allowable expressions take the following forms:

```
register in constant .. constant
register not in constant .. constant

memory in constant .. constant
memory not in constant .. constant
```

Here is a simple example of the first form with the code that HLA generates for the expression:

```
if( eax in 1..10 ) then

    mov( 1, ebx );

endif;
```

Resulting (MASM) assembly code:

```
                cmp     eax, 1
                jb      ?1_false
                cmp     eax, 10
                ja      ?1_false
                mov     ebx, 1
?1_false:
```

Once again, you can see that HLA generates reasonable assembly code without modifying any register values. Note that if modifying the EAX register is okay, you can write slightly better code by using the following sequence:

```
                dec     eax
                cmp     eax, 9
                ja      ?1_false
                mov     ebx, 1
?1_false:
```

While, in general, a simplification like this is not possible you should always remember how HLA generates code for the range comparisons and decide if it is appropriate for the situation.

By the way, the "not in" form of the range comparison does generate slightly different code that the form above. Consider the following:


```
    if( eax not in 1..10 ) then

        mov( 1, eax );

    endif;
```

HLA generates the following (MASM) assembly language code for the sequence above:

```
                cmp     eax, 1
                jb      ?2_true
                cmp     eax, 10
                jna     ?1_false
?2_true:
                mov     eax, 1
?1_false:
```

As you can see, though the code is slightly different it is still exactly what you would probably write if you were writing the low level code yourself.

HLA also allows a limited form of the boolean expression that checks to see if a character value in an eight-bit register is a member of a character set constant or variable. These expressions use the following general syntax:

reg$_8$ in *CSet_Constant*
reg$_8$ in *CSet_Variable*

reg$_8$ not in *CSet_Constant*
reg$_8$ not in *CSet_Variable*

These forms were included in HLA because they are so similar to the range comparison syntax. However, the code they generate may not be particularly efficient so you should avoid using these expression forms if code speed and size need to be optimal. Consider the following:

```
    if( al in {'A'..'Z','a'..'z', '0'..'9'} ) then

        mov( 1, eax );

    endif;
```

This generates the following (MASM) assembly code:

```
strings         segment page public 'data'
?1_cset          byte 00h,00h,00h,00h,00h,00h,0ffh,03h
                 byte 0feh,0ffh,0ffh,07h,0feh,0ffh,0ffh,07h
strings         ends

                push    eax
                movzx   eax, al
                bt      dword ptr ?1_cset, eax
                pop     eax
                jnc     ?1_false
                mov     eax, 1
?1_false:
```


This code is rather lengthy because HLA never assumes that it cannot disturb the values in the CPU registers. So right off the bat this code has to push and pop EAX since it disturbs the value in EAX. Next, HLA doesn't assume that the upper three bytes of EAX already contain zero, so it zero fills them. Finally, as you can see above, HLA has to create a 16-byte character set in memory in order to test the value in the AL register. While this is convenient, HLA does generate a lot of code and data for such a simple looking expression. Hence, you should be careful about using boolean expressions involving character sets if speed and space is important. At the very least, you could probably reduce the code above to something like:

```
                movzx( charToTest, eax );
                bt( eax, {'A'..'Z','a'..'z', '0'..'9'});
                jnc SkipMov;
                mov(1, eax );
SkipMov:
```

This generates code like the following:

```
strings         segment page public 'data'
?cset_3          byte   00h,00h,00h,00h,00h,00h,0ffh,03h
                 byte   0feh,0ffh,0ffh,07h,0feh,0ffh,0ffh,07h
strings         ends

                movzx   eax, byte ptr ?1_charToTest[0]  ;charToTest
                bt      dword ptr ?cset_3, eax
                jnc     ?4_SkipMov
                mov     eax, 1

?4_SkipMov:
```

As you can see, this is slightly more efficient. Fortunately, testing an eight-bit register to see if it is within some character set (other than a simple range, which the previous syntax handles quite well) is a fairly rare operation, so you generally don't have to worry about the code HLA generates for this type of boolean expression.

HLA lets you specify a register name or a memory location as the only operand of a boolean expression. For registers, HLA will use the TEST instruction to see if the register is zero or non-zero. For memory locations, HLA will use the CMP instruction to compare the memory location's value against zero. In either case, HLA will emit a JNE or JE instruction to branch around the code to skip (e.g., in an IF statement) if the result is zero or non-zero (depending on the form of the expression).

register
!*register*

memory
!*memory*

You should not use this trick as an efficient way to test for zero or not zero in your code. The resulting code is very confusing and difficult to follow. If a register or memory location appears as the sole operand of a boolean expression, that register or memory location should hold a boolean value (true or false). Do not think that "if(eax) then..." is any more efficient than "if(eax<>0) then..." because HLA will actually emit the same exact code for both statements (i.e., a TEST instruction). The second is a lot easier to understand if you're really checking to see if EAX is not zero (rather than it contains the boolean value true), hence it is always preferable even if it involves a little extra typing.

Example:


```
    if( eax != 0 ) then

        mov( 1, ebx );

    endif;

    if( eax ) then

        mov( 2, ebx );

    endif;
```

The code above generates the following assembly instruction sequence:

```
                test    eax,eax ;Test for zero/false.
                je      ?2_false
                mov     ebx, 1
?2_false:
                test    eax,eax ;Test for zero/false.
                je      ?3_false
                mov     ebx, 2
?3_false:
```

Note that the pertinent code for both sequences is identical. Hence there is never a reason to sacrifice readability for efficiency in this particular case.

The last form of boolean expression that HLA allows is a flag designation. HLA uses symbols like @c, @nc, @z, and @nz to denote the use of one of the flag settings in the CPU FLAGS register. HLA supports the use of the following flag names in a boolean expression:

```
@c, @nc, @o, @no, @z, @nz, @s, @ns,
@a, @na, @ae, @nae, @b, @nb, @be, @nbe,
@l, @nl, @g, @ne, @le, @nle, @ge, @nge,
@e, @ne
```

Whenever HLA encounters a flag name in a boolean expression, it efficiently compiles the expression into a single conditional jump instruction. So the following IF statement's expression compiles to a single instruction:

```
if( @c ) then

    << do this if the carry flag is set >>

endif;
```

The above code is completely equivalent to the sequence:

```
    jnc SkipStmts;

    << do this if the carry flag is set >>

SkipStmts:
```

The former version, however, is more readable so you should use the IF form wherever practical.

L.5 The JT/JF Pseudo-Instructions

The JT (jump if true) and JF (jump if false) pseudo-instructions take a boolean expression and a label. These instructions compile into a conditional jump instruction (or sequence of instructions) that jump to the target label if the specified boolean expression evaluates false. The compilation of these two statements is almost exactly as described for boolean expressions in the previous section.

The following are a couple of examples that show the usage and code generation for these two statements.

```
lbl2:
    jt( eax > 10 ) label;
label:
    jf( ebx = 10 ) lbl2;


; Translated Code:

?2_lbl2:
       cmp eax, 10
       ja ?4_label

?4_label:

       cmp ebx, 10
       jne ?2_lbl2
```

L.6 The HLA if..then..elseif..else..endif Statement, Part II

With the discussion of boolean expressions out of the way, we can return to the discussion of the HLA IF statement and expand on the material presented earlier. There are two main topics to consider: the inclusion of the ELSEIF and ELSE clauses and the HLA hybrid IF statement. This section will discuss these additions.

The ELSE clause is the easiest option to describe, so we'll start there. Consider the following short HLA code fragment:

```
    if( eax < 10 ) then

        mov( 1, ebx );

    else

        mov( 0, ebx );

    endif;
```

HLA's code generation algorithm emits a JMP instruction upon encountering the ELSE clause; this JMP transfers control to the first statement following the ENDIF clause. The other difference between the IF/ELSE/ENDIF and the IF/ENDIF statement is the fact that a false expression evaluation transfers control to the ELSE clause rather than to the first statement following the ENDIF. When HLA compiles the code above, it generates machine code like the following:


```
                cmp     eax, 10
                jnb     ?2_false   ;Branch to ELSE section if false

                mov     ebx, 1
                jmp     ?2_endif   ;Skip over ELSE section

; This is the else section:

?2_false:
                mov     ebx, 0
?2_endif:
```

About the only way you can improve upon HLA's code generation sequence for an IF/ELSE statement is with knowledge of how the program will operate. In some rare cases you can generate slightly better performing code by moving the ELSE section somewhere else in the program and letting the THEN section fall straight through to the statement following the ENDIF (of course, the ELSE section must jump back to the first statement after the ENDIF if you do this). This scheme will be slightly faster if the boolean expression evaluates true most of the time. Generally, though, this technique is a bit extreme.

The ELSEIF clause, just as its name suggests, has many of the attributes of an ELSE and and IF clause in the IF statement. Like the ELSE clause, the IF statement will jump to an ELSEIF clause (or the previous ELSEIF clause will jump to the current ELSEIF clause) if the previous boolean expression evaluates false. Like the IF clause, the ELSEIF clause will evaluate a boolean expression and transfer control to the following ELSEIF, ELSE, or ENDIF clause if the expression evaluates false; the code falls through to the THEN section of the ELSEIF clause if the expression evaluates true. The following examples demonstrate how HLA generates code for various forms of the IF..ELSEIF.. statement:

Single ELSEIF clause:

```
    if( eax < 10 ) then

        mov( 1, ebx );

    elseif( eax > 10 ) then

        mov( 0, ebx );

    endif;


; Translated code:

                cmp     eax, 10
                jnb     ?2_false
                mov     ebx, 1
                jmp     ?2_endif
?2_false:
                cmp     eax, 10
                jna     ?3_false
                mov     ebx, 0
?3_false:
?2_endif:
```

Single ELSEIF clause with an ELSE clause:


```
    if( eax < 10 ) then

        mov( 1, ebx );

    elseif( eax > 10 ) then

        mov( 0, ebx );

    else

        mov( 2, ebx );

    endif;


; Converted code:

                cmp     eax, 10
                jnb     ?2_false
                mov     ebx, 1
                jmp     ?2_endif
?2_false:
                cmp     eax, 10
                jna     ?3_false
                mov     ebx, 0
                jmp     ?2_endif
?3_false:
                mov     ebx, 2
?2_endif:
```

IF statement with two ELSEIF clauses:

```
    if( eax < 10 ) then

        mov( 1, ebx );

    elseif( eax > 10 ) then

        mov( 0, ebx );

    elseif( eax = 5 ) then

        mov( 2, ebx );

    endif;

; Translated code:

                cmp     eax, 10
                jnb     ?2_false
                mov     ebx, 1
                jmp     ?2_endif
?2_false:
                cmp     eax, 10
                jna     ?3_false
                mov     ebx, 0
                jmp     ?2_endif
```



```
?3_false:
                mov     ebx, 2
?2_endif:
```

IF statement with two ELSEIF clauses and an ELSE clause:

```
    if( eax < 10 ) then

        mov( 1, ebx );

    elseif( eax > 10 ) then

        mov( 0, ebx );

    elseif( eax = 5 ) then

        mov( 2, ebx );

    else

        mov( 3, ebx );

    endif;

; Translated code:

                cmp     eax, 10
                jnb     ?2_false
                mov     ebx, 1
                jmp     ?2_endif
?2_false:
                cmp     eax, 10
                jna     ?3_false
                mov     ebx, 0
                jmp     ?2_endif
?3_false:
                cmp     eax, 5
                jne     ?4_false
                mov     ebx, 2
                jmp     ?2_endif
?4_false:
                mov     ebx, 3
?2_endif:
```

This code generation algorithm generalizes to any number of ELSEIF clauses. If you need to see an example of an IF statement with more than two ELSEIF clauses, feel free to run a short example through the HLA compiler to see the result.

In addition to processing boolean expressions, the HLA IF statement supports a hybrid syntax that lets you combine the structured nature of the IF statement with the unstructured nature of typical assembly language control flow. The hybrid form gives you almost complete control over the code generation process without completely sacrificing the readability of an IF statement. The following is a typical example of this form of the IF statement:


```
    if
    (#{
        cmp( eax, 10 );
        jna false;
    }#) then

        mov( 0, eax );

    endif;


; The above generates the following assembly code:

                cmp     eax, 10
                jna     ?2_false
?2_true:
                mov     eax, 0
?2_false:
```

Of course, the hybrid IF statement fully supports ELSE and ELSEIF clauses (in fact, the IF and ELSEIF clauses can have a potpourri of hybrid or traditional boolean expression forms). The hybrid forms, since they let you specify the sequence of instructions to compile, put the issue of efficiency squarely in your lap. About the only contribution that HLA makes to the inefficiency of the program is the insertion of a JMP instruction to skip over ELSEIF and ELSE clauses.

Although the hybrid form of the IF statement lets you write very efficient code that is more readable than the traditional "compare and jump" sequence, you should keep in mind that the hybrid form is definitely more difficult to read and comprehend than the IF statement with boolean expressions. Therefore, if the HLA compiler generates reasonable code with a boolean expression then by all means use the boolean expression form; it will probably be easier to read.

L.7 The While Statement

The only difference between an IF statement and a WHILE loop is a single JMP instruction. Of course, with an IF and a JMP you can simulate most control structures, the WHILE loop is probably the most typical example of this. The typical translation from WHILE to IF/JMP takes the following form:

```
while( expr ) do

    << statements >>

endwhile;


// The above translates to:

label:
    if( expr ) then

          << statements >>
          jmp label;

    endif;
```


Experienced assembly language programmers know that there is a slightly more efficient implementation if it is likely that the boolean expression is true the first time the program encounters the loop. That translation takes the following form:

```
    jmp testlabel;
label:

    << statements >>

testlabel:
    JT( expr ) label;   // Note: JT means jump if expression is true.
```

This form contains exactly the same number of instructions as the previous translation. The difference is that a JMP instruction was moved out of the loop so that it executes only once (rather than on each iteration of the loop). So this is slightly more efficient than the previous translation. HLA uses this conversion algorithm for WHILE loops with standard boolean expressions.

L.8 repeat..until

L.9 for..endfor

L.10 forever..endfor

L.11 break, breakif

L.12 continue, continueif

L.13 begin..end, exit, exitif

L.14 foreach..endfor

L.15 try..unprotect..exception..anyexception..endtry, raise

Editorial Note: This document is a work in progress. At some future date I will finish the sections above. Until then, use the HLA "-s" compiler option to emit MASM code and study the MASM output as described in this appendix.

