The Quick Guide to HLA

1 Overview

This guide is designed to help those who are already familiar with x86 assembly language programming to get up to speed with HLA as rapidly as possible. HLA was designed as a tool for teaching assembly language programming to University/College students who have no prior experience with assembly language but have some high level language programming experience (C/C++, Pascal, Java, etc.). The documentation that exists for HLA comes in two forms: the HLA reference manuals and the "Art of Assembly Language Programming/32-bit Edition." The "Art of Assembly" text is suitable for students and beginners to assembly language programming; it starts from square one and teaches assembly language programming using HLA. Unfortunately, this text is not particularly suitable for those programmers who already know assembly language. The HLA reference manuals are great when you need to look up some particular feature. They do fully explain the HLA language, however, the HLA language is rather large so the assembly programmer who is new to HLA is faced with reading a tremendous amount of material just to get started with HLA. Most individuals won't bother. The purpose of this guide is to present a very small subset of HLA to the advanced x86 assembly language programmer in as few pages as possible. This guide does not attempt to teach any of HLA's special features; it assumes the reader is using an assembler such as MASM, TASM, NASM, Gas, etc., and is interested in learning how to write assembly code using HLA in a fashion comparable to those assemblers. Of course, the whole reason for such a person to learn HLA is to be able to take advantage of HLA's advanced features. However, one has to learn to walk before they run, this is the guide that will get that person walking. Once the reader is comfortable using HLA in a "traditional assembly" sense, then that reader can refer to the HLA reference manuals in order to learn the more advanced features of the language.

2 Installing HLA

This manual assumes that you've already installed HLA on your system and you've verified its operation. Please see the HLA Reference Guide for details on installating HLA on your system.

3 Running HLA

HLA is a command line tool that you run from the Win32 Command Prompt. This document assumes that you are familiar with basic command prompt syntax and you're familiar with various commands like "DIR" and "RENAME". To run HLA from the command line prompt, you use a command like the following:

```
hla optional_command_line_parameters Filename_list
```

The filename list consists of one or more unambiguous filenames having the extension: HLA, ASM, or OBJ. HLA will first run the HLAPARSE program on all files with the HLA extension (producing files with the same basename and an ASM or INC extension). Then HLA runs the MASM program on all files with the ASM extension (including the files produced by HLAPARSE). Finally, HLA runs the linker to combine all the OBJ files together (including the OBJ files that MASM produces). The ultimate result, assuming there were no errors along the way, is an executable file with an EXE extension.

HLA supports the following command line parameters:

```
-@       Do not generate a linker response file.
-aXXXXX  Pass XXXXX to assembler as command line option.
-c       Compile and assemble to .OBJ files only.
-dXXXXX  Define symbol XXXXX and set it to true during compile.
-e name  Executable output filename.
-lXXXXX  Pass XXXXX to linker as command line option.
-m       Create a map file during link.
-masm    Use MASM to assemble output.
-o:omf   When using MASM, produce OMF output rather than COFF.
-o:win32 Produce Win32 COFF files.
-s       Compile to .ASM files only.
-sm      Compile to MASM-compatible .ASM file (default).
-st      Compile to TASM-compatible .ASM file.
-sym     Dump symbol table after compile.
-tasm    Assemble file using TASM32.EXE assembler.
-test    Send diagnostic info to stdout rather than stderr
          (This option is intended for HLA test/debug purposes).
-v       Verbose compile.
-w       Compile as windows app (default is console app).
-?       Display this help message.
```

Please see the HLA Reference Manual for an explanation of each of these options. Most of the time, you will not use any of these options when compiling typical HLA programs. The "-c" and "-s" options are the ones you will use most commonly (and this document assumes that you understand their purpose).

4 HLA Language Elements

Starting with this section we being discussing the HLA source language. HLA source files must contain only seven-bit ASCII characters. These are Windows text files with each source line record containing a carriage return/line feed termination sequence. White space consists of spaces, tabs, and newline sequences. Generally, HLA does not appreciate other control characters in the file and may generate an error if they appear in the source file.

4.1 Comments

HLA uses "//" to lead off single line comments. It uses "/*" to begin an indefinite length comments and it uses "*/" to end an indefinite length comment. C/C++, Java, and Delphi users will be quite comfortable with this notation.

4.2 Special Symbols

The following characters are HLA lexical elements and have special meaning to HLA:

```
*  /  +  -  (  )  [  ]  {  }  <  >  :  ;  ,  .  =  ? & | ^ ! @
&&    ||    <=    >=     <>    !=    ==    :=    ..    <<    >>
##    #(    )#    #{    }#
```

This document will not explain the meaning of all these symbols, only the minimum necessary to write simple HLA programs. See the HLA Reference Manual for more details.

4.3 Reserved Words

HLA supports a large number of reserved words (mostly, they are machine instructions). For brevity, that list does not appear here; please see the HLA reference manual for a complete and up-to-date list. Note that HLA does not allow you to use a reserved word as a program identifier, so you should scan over the list at least once to familiarize yourself with reserved words that you might be used to using as identifiers in your assembly language programs. HLA reserved words are case insensitive. That is, "MOV" and "mov" (as well as any permutation with resepect to case) both represent the HLA "mov" reserved word.

4.4 External Symbols and Assembler Reserved Words

HLA produces an assembly language file during compilation and invokes an assembler such as MASM, TASM, or Gas to complete the compilation process. HLA automatically translates normal identifiers you declare in your program to beneign identifiers in the assembly language program. However, HLA does not translate EXTERNAL symbols, but preserves these names in the assembly language file it produces. Therefore, you must take care not to use external names that conflict with the underlying assembler's set of reserved words or that assembler will generate an error when it attempts to process HLA's output.

For a list of assembler reserved words, please see the documentation for the assembler you are using to assembly HLA's output (i.e., MASM, TASM, or Gas).

4.5 HLA Identifiers

HLA identifiers must begin with an alphabetic character or an underscore. After the first character, the identifier may contain alphanumeric and underscore symbols. There is no technical limit on identifier length in HLA, but you should avoid external symbols greater than about 32 characters in length since the assembler and linkers that process HLA identifiers may not be able to handle such symbols.

HLA identifiers are always *case neutral*. This means that identifiers are case sensitive insofar as you must always spell an identifier exactly the same (with respect to alphabetic case). However, you are not allowed to declare two identifiers whose only difference is alphabetic case.

4.6 External Identifiers

HLA lets you explicitly provide a string for external identifiers. External identifiers are not limited to the format for HLA identifiers. HLA allows any string constant to be used for an external identifier. It is your responsibility to use only those characters that are legal in the assembler that processes HLA's intermediate ASM file. Note that this feature lets you use symbols that are not legal in HLA but are legal in external code (e.g., Win32 APIs use the '@' character in identifiers). See the discussion of the @EXTERNAL option for more details.

4.7 Data Types in HLA

4.7.1 Native (Primitive) Data Types in HLA

HLA provides the following basic primitive types:

```
One-byte types: byte, boolean, enum, uns8, int8, and char.
Two-byte types: word, uns16, int16.
Four-byte types: dword, uns32, int32, real32, string, pointer
Eight-byte types: uns64, int64, qword, thunk, and real64.
Ten-Byte types: tbyte, and real80.
Sixteen-byte types: uns128, int128, lword, and cset
```

For details on these particular types, please consult the HLA Reference Manual. This document will make use of the following types:

```
byte, word, dword, string, real32, qword, real64, and real80
```

These are the types a typical assembler provides.

BYTE variables and objects may hold integer numeric values in the range -128..+255, any ASCII character constant, and the two predefined boolean values *true* (1) and *false* (0). Normally, HLA does a small amount of type checking; however, you can associate any value that can fit into eight bits with a byte-sized variable (or other object).

WORD variables and object may hold integer numeric values in the range -32768..+65535. Generally, HLA does not allow the association of other values with a WORD object.

DWORD variables and objects may hold integer numeric values in the range -2147483647..+4294967295, or the address of an object (using the "&" address-of operator).

STRING variables are also DWORD objects. STRING objects hold the address of a sequence of zero or more ASCII characters that end with a zero byte. In the four bytes immediately preceding the location contained in the string pointer is the current length of the string. In the four bytes preceding the current length is the maximum allowable length of the string. Note that HLA strings are "read-only" compatible

with ASCIIZ strings used by Windows and C/C++ (read-only meaning that you can pass an HLA string to a Windows API or C/C++ function but that function should not modify the string).

QWORD, UNS64, and INT64 objects consume eight bytes of memory. As of HLA v1.37, support is available for 64-bit unsigned, signed, and hexadecimal numeric constants. TBYTE objects consume ten bytes (80 bits), but, again, HLA does not support the use of 80-bit integer or BCD constants in a program. LWORD, UNS128, and INT128 values are also legal and support 128-bit hexadecimal, unsigned, or signed constants.

REAL32, REAL64, and REAL80 types in HLA support the three different IEEE floating point formats.

4.8 Composite Data Types

In addition to the primitive types above, HLA supports arrays, records (structures), unions, classes, and pointers of the above types (except for text objects).

4.8.1 Array Data Types

HLA allows you to create an array data type by specifying the number of array elements after a type name. Consider the following HLA type declaration that defines intArray to be an array of dword objects:

```
type intArray : dword[ 16 ];
```

The "[16]" component tells HLA that this type has 16 four-byte double words. HLA arrays use a zero-based index, so the first element is always element zero. The index of the last element, in this example, is 15 (total of 16 elements with indicies 0..15).

HLA also supports multidimensional arrays. You can specify multidimensional arrays by providing a list of indicies inside the square brackets, e.g.,

```
type intArray4x4 : dword[ 4, 4 ];
type intArray2x2x4 : dword[ 2,2,4 ];
```

4.8.2 Record Data Types[1]

HLA's records allow programmers to create data types whose fields can be different types. The following HLA static variable declaration defines a simple record with four fields:

```
static Planet: record

                x:          dword;
                y:          dword;
                z:          dword;
                density:    real64;

            endrecord;
```

Objects of type Planet will consume 20 bytes of storage at run-time.

The fields of a record may be of any legal HLA data type including other composite data types. You use dot-notation to access fields of a record object, e.g.,

1. For C/C++ programmers: an HLA record is similar to a C struct. In language design terminology, a record is often referred to as a "cartesian product."

```
mov( Planet.x, eax );
```

4.9 Literal Constants

Literal constants are those language elements that we normally think of as non-symbolic constant objects. HLA supports a wide variety of literal constants. The following sections describe those constants.

4.9.1 Numeric Constants

HLA lets you specify several different types of numeric constants.

Decimal Constants:

The first and last characters of a decimal integer constant must be decimal digits (0..9). Interior positions may contain decimal digits and underscores. The purpose of the underscore is to provide a better presentation for large decimal values (i.e., use the underscore in place of a comma in large values). Example: 1_234_265.

Hexadecimal Constants:

Hexadecimal literal constants must begin with a dollar sign (“$”) followed by a hexadecimal digit and must end with a hexadecimal digit (0..9, A..F, or a..f). Interior positions may contain hexadecimal digits or underscores. Hexadecimal constants are easiest to read if each group of four digits (starting from the least significant digit) is separated from the others by an underscore. E.g., $1A_2F34_5438.

Binary Constants:

Binary literal constants begin with a percent sign (“%”) followed by at least one binary digit (0/1) and they must end with a binary digit. Interior positions may contain binary digits or underscore characters. Binary constants are easiest to read if each group of four digits (starting from the least significant digit) is separated from the others by an underscore. E.g., %10_1111_1010.

Real (Floating Point) Constants:

Floating point (real) literal constants always begin with a decimal digit (never just a decimal point). A string of one or more decimal digits may be optionally followed by a decimal point and zero or more decimal digits (the fractional part). After the optional fractional part, a floating point number may be followed by “e” or “E”, a sign (“+” or “-”), and a string of one or more decimal digits (the exponent part). Underscores may appear between two adjacent digits in the floating point number; their presence is intended to substitute for commas found in real-world decimal numbers.

Boolean Constants:

Boolean constants consist of the two predefined identifiers *true* and *false*. Note that your program may redefine these identifiers, but doing so is incredibly bad programming style.

Character Constants:

Character literals generally consist of a single (graphic) character surrounded by apostrophes. To represent the apostrophe character, use four apostrophies, e.g., ‘””’.

Another way to specify a character constant is by typing the “#” symbol followed by a numeric literal constant (decimal, hexadecimal, or binary). Examples: #13, #$D, #%1101.

String Constants:

String literal constants consist of a sequence of (graphic) characters surrounded by quotes. To embed a quote within a string, insert a pair of quotes into the string, e.g., “He said ““This”” to me.”

If two string literal constants are adjacent in a source file (with nothing but whitespace between them), then HLA will concatenate the two strings and present them to the parser as a single string. Furthermore, if a character constant is adjacent to a string, HLA will concatenate the character and string to form a single string object. This is useful, for example, when you need to embed control characters into a string, e.g.,

```
"This is the first line" #$d #$a "This is the second line" #$d #$a
```

HLA treats the above as a single string with a newline sequence (CR/LF) at the end of each of the two lines of text.

Pointer Constants:

HLA allows a very limited form of a pointer constant. If you place an ampersand in front of a static object's name (i.e., the name of a static variable, readonly variable, uninitialized variable, segment variable, procedure, method, or iterator), HLA will compute the run-time offset of that variable. Pointer constants may not be used in abitrary constant expressions. You may only use pointer constants in expressions used to initialize static or readonly variables or as constant expressions in 80x86 instructions.

Structured Constants:

HLA also supports certain structured constants including character set constants, array constants, and record constants. Please see the HLA Reference Manual for more details.

4.10 Constant Expressions in HLA

HLA provides a rich expression evaluator to process assembly-time expressions. HLA supports the following operators (sorting by decreasing precedence):

```
! (unary not),- (unary negation)
*, div, mod, /, <<, >>
+, -
=, = =, <>, !=, <=, >=, <, >
&, |, &, in
```

!*expr*

The expression must be either boolean or a number. For boolean values, *not* ("!") computes the standard logical not operation. For numbers, *not* ("!") computes the bitwise not operation on the bits of the number.

```
- expr           (unary negation operator)
expr1 * expr2    (multiplication operator)
expr1 div expr2  (integer division operator)
expr1 mod expr2  (integer remainder operator)
expr1 / expr2    (real division operator)
expr1 << expr2   (integer shift left operator)
expr1 >> expr2   (integer shift right operator)
expr1 + expr2    (addition operator)
expr1 - expr2    (subtraction operator)
expr1 = expr2    (equality comparison operator)
expr1 <> expr2   (inequality comparison operator)
expr1 < expr2    (less than comparison operator)
expr1 <= expr2   (less than or equal comparison operator)
expr1 > expr2    (greater than comparison operator)
expr1 >= expr2   (greater or equal comparison operator)
expr1 & expr2    (logical/boolean AND operator)
expr1 | expr2    (logical/boolean OR operator)
expr1 ^ expr2    (logical/boolean XOR operator)
( expr )         (override operator precedence)
```

HLA supports several other constant operators. Furthermore, many of the above operators are overloaded depending on the operand types. Note that for numeric (integer) operands, HLA fully support 128-bit arithmetic. Please see the HLA Reference Manual for more details.

4.11 Program Structure

An HLA program uses the following general syntax:

```
program identifier ;
    declarations
begin identifier;
    statements
end identifier;
```

The three identifiers above must all match. The declaration section (declarations) consists of type, const, val, var, static, uninitialized, readonly, segment, procedure, and macro definitions. Any number of these sections may appear and they may appear in any order; more than one of each section may appear in the declaration section.

If you wish to write a library module that contains only procedures and no main program, you would use an HLA unit. Units have a syntax that is nearly identical to programs, there isn't a BEGIN associated with the unit, e.g.,

```
unit TestPgm;

    procedure LibraryRoutine;
    begin LibraryRoutine;
       << etc. >>
    end LibraryRoutine;

end TestPgm;
```

4.11.1 Procedure Declarations

Procedure declarations are nearly identical to program declarations.

```
procedure identifier;  noframe;
begin identifier;
    statements
end identifier;
```

Note that HLA procedures provide a very rich set of syntactical options. The template above corresponds to the syntax that creates procedures most closely resembling those that other assemblers use. HLA's procedures allow parameters, local variable declarations, and lots of other features this document won't describe. For more details, please see the HLA Reference Manual.

Note, *and this is very important*, that the procedure option NOFRAME must appear in the PROCEDURE declaration. Without this declaration, HLA inserts some additional code into your procedure and it will probably fail to work as you intend (indeed, it's likely the inserted code will crash when it runs).

Example of a procedure:

```
procedure ProcDemo; @noframe;
begin ProcDemo;

    add( 5, eax );
    ret();

end ProcDemo;
```

4.12 Declarations

Programs, units, procedures, methods, and iterators all have a declaration section. Classes and namespaces also have a declaration section, though it is somewhat limited. A declaration section can contain one or more of the following components (among other things this document doesn't cover):

- A type section.
- A const section.
- A static section.
- A procedure.

The order of these sections is irrelevant as long as you ensure that all identifiers used in a program are defined before their first use. Furthermore, as noted above, you may have multiple sections within the same set of declarations. For example, the two const sections in the following procedure declaration are perfectly legal:

```
program TwoConsts;
const   MaxVal := 5;
type    Limits: dword[ MaxVal ];
const   MinVal := 0;
begin TwoConsts;

    //...

end TwoConsts;
```

4.12.1 Type Section

You can declare user-defined data types in the type section. The type section appears in a declaration section and begins with the reserved word **type**. It continues until encountering another declaration reserved word (e.g., const, var, or val) or the reserved word **begin**. A typical type definition begins with an indentifier followed by a colon and a type definition. The following paragraphs describe some of the legal types of type definitions.

```
id1 : id2;   // Defines id1 to be the same as type id2.
id1 : id2 [ dim_list ];  // Defines id1 to be an array of type id2.
id1 : record     // Defines id1 as a record type.
      field_declarations
   endrecord;
```

4.12.2 Const Section

You may declare manifest constants in the CONST section of an HLA program. It is illegal to attempt to change the value of a constant at some later point during assembly. Of course, at run-time the constant always has a static value.

The constant declaration section begins with the reserved word **const** and is followed by a sequence of constant definitions. The constant declaration section ends when HLA encounters a keyword such as **const, type**, **var**, **val**, etc. Actual constant definitions take the forms specified in the following paragraphs.

```
id := expr;  // Assigns the value and type of expr to id
id1 : id2 := expr;  // Creates constant id1 of type id2 of value expr.
```

Note that HLA supports several types of constants this section doesn't discuss (e.g., array and record constants and well as compile-time variables). See the HLA Reference Manual for more details.

4.12.3 Static Section

The **static** section lets you declare static variables you can reference at run-time by your code. The following paragraphs list some of the forms that are legal in the STATIC section. As usual, see the HLA Reference Manual for lots of additional features that HLA supports in the STATIC section.

```
static
    id1 : id2;          // Declares variable id1 of type id2
    id1 : id2 := expr;  // Declares variable id1 of type id2, init'd with expr
    id1 : id2[ expr ];  // Declares array id1 of type id2 with expr elements
```

4.12.3.1 The @NOSTORAGE Option

The @nostorage option tells HLA to associate the current offset in the segment with the specified variable, but don't actually allocate any storage for the object. This option effectively creates an alias of the current variable with the next object you declare in one of the static sections. Consider the following example:

```
static
    b:      byte; @nostorage;
    w:      word; @nostorage;
    d:      dword;
```

Because the b and w variables both have the @nostorage option associated with them, HLA does not reserve any storage for these variables. The d variable does not have the nostorage option, so HLA does reserve four bytes for this variable. The b and w variables, since they don't have storage associated with them, share the same address in memory with the d variable.

4.12.3.2 The @EXTERNAL Option

The external option gives you the ability to reference static variables that you declare in other files. Like the external clause for procedures, there are two different syntaxes for the external clause appearing after a variable declaration:

```
varName: varType; @external;
varName: varType; @external( "external_Name" );
```

The first form above uses the variable's name for both the internal and external names. The second form uses varName as the internal name that HLA uses and it associates this varible with external_Name in the external modules. The @external option is always the last option associated with a variable declaration.

If the actual variable definition for an external object appears in a source file after an external declaration, this tells HLA that the definition is a public variable that other modules may access (the default is local to the current source file). This is the only way to declare a variable to be public so that other modules can use it. Usually, you would put the external declaration in a header file that all modules (wanting to access the variable) include; you also include this header file in the source file containing the actual variable declaration.

4.12.4 Macros

HLA has one of the most powerful macro expansion facilities of any programming language. HLA's macros are the key to extending the HLA language. If you're a big user of macros then you will want to read the HLA Reference Manual to learn all about HLA's powerful macro facilities. This section will describe HLA's limited "Standard Macro" facility which is comparable to the macro facilities other assemblers provide.

You can declare macros in the declaration section of a program using the following syntax:

```
#macro identifier ( optional_parameter_list ) ;
    statements
#endmacro;
```

Example:

```
#macro MyMacro;
    ?i = i + 1;
#endmacro;
```

The optional parameter list must be a list of one or more identifiers separated by commas. HLA automatically associates the type "text" with all macro parameters (except for one special case noted below). Example:

```
#macro MacroWParms( a, b, c );
    ?a = b + c;
#endmacro;
```

If the macro does not allow any parameters, then you follow the identifier with a semicolon (i.e., no parentheses or parameter identifiers). See the first example in this section for a macro without any parameters.

Occasionally you may need to define some symbols that are local to a particular macro invocation (that is, each invocation of the macro generates a unique symbol for a given identifier). The local identifier list allows you to do this. To declare a list of local identifiers, simply following the parameter list (after the parenthesis but before the semicolon) with a colon (":") and a comma separated list of identifiers, e.g.,

```
#macro ThisMacro(parm1):id1,id2;
...
```

HLA automatically renames each symbol appearing in the local identifier list so that the new name is unique throughout the program. HLA creates unique symbols of the form "_XXXX_" where XXXX is some hexadecimal numeric value. To guarantee that HLA can generate unique symbols, you should avoid defining symbols of this form in your own programs (in general, symbols that begin and end with an underscore are reserved for use by the compiler and the HLA standard library). Example:

```
#macro LocalSym : i,j;

j:  cmp(ax, 0)
    jne( i )
    dec( ax )
    jmp( j )
i:
#endmacro;
```

To invoke a macro, you simply supply its name and an appropriate set of parameters. Unless you specify a variable number of parameters (using the array syntax) then the number of actual parameters must exactly match the number of formal parameters. If you specify a variable number of parameters, then the number of actual parameters must be greater than or equal to the number of formal parameters (not counting the array parameter).

Actual macro parameters consist of a string of characters up to, but not including a separate comma or the closing parentheses, e.g.,

```
example( v1, x+2*y )
```

"v1" is the text for parameter #1, "x+2*y" is the text for parameter #2. Note that HLA strips all leading whitespace and control characters before and after the actual parameter when expanding the code in-line. The example immediately above would expand do the following:

```
?v1 := x+2*y;
```

If (balanced) parentheses appear in some macro's actual parameter list, HLA does not count the closing parenthesis as the end of the macro parameter. That is, the following is perfectly legal:

```
example(  v1, ((x+2)*y) )
```

This expands to:

```
?v1 := ((x+2)*y);
```

4.12.5 The #Include Directive

Like most languages, HLA provides a source inclusion directive that inserts some other file into the middle of a source file during compilation. HLA's #INCLUDE directive is very similar to the pragma of the same name in C/C++ and you primarily use them both for the same purpose: including library header files into your programs.

HLA's include directive has the following syntax:

```
#include( string_expression );
```

4.12.6 The #IncludeOnce Directive

When composing complex header files, particularly when constructing library header files, you may find in necessary to insert a #INCLUDE("file") directive into some other header files. Generally, this is not a problem, HLA certainly allows nested include files (up to 256 files deep). However, unless you are very careful about how you organize your files, it is very easy to create an "include loop" where one header file includes another and that other header file includes the first. Attempting to compile a program that includes either header file results in an infinite "include loop" during compilation. Clearly, this is not desirable.

You can use the #INCLUDEONCE directive to avoid this problem. The only difference between the two is that HLA keeps track of all files it has processed using the #INCLUDE or #INCLUDEONCE directives and will not process a header file a second time if you attempt to include it using the #INCLUDEONCE directive.

Whenever HLA processes the #INCLUDEONCE directive, it first compares its string operand with a list of strings appearing in previous #INCLUDE or #INCLUDEONCE directives. If it matches one of these previous strings, then HLA ignores the #INCLUDEONCE directive; if the include filename does not appear in HLA' internal list, then HLA adds this filename to the list and includes the file.

4.12.7 The #asm..#endasm and #emit Directives

HLA is far from perfect. There are many missing instructions (some left out on purpose, some left out because of laziness, and some left out because of ignorance). For example, HLA currently doesn't support the SSE instruction set. Fret not, though, HLA v1.x provides two escape mechanisms that let you do anything legal in MASM or the underlying assembler used to process HLA's output.

The first of these escape mechanisms is the #ASM..#ENDASM section. The syntax for an assembly block is as follows:

```
#asm
    << text that is     >>
    << emitted directly >>
    << to the MASM out- >>
    << put file.        >>
#endasm
```

All text appearing between the #ASM and #ENDASM directives is emitted directly to the .ASM output file produced by HLA. MASM (or whatever assembler you're using) will be responsible for assembling this code. Of course, you must supply source code that is compatible with your assembler in an assembly block or the assembly process will fail when the assembler encounters your code.

The second escape mechanism is the #EMIT directive. This directive takes the following form:

```
#emit( string_expression )
```

HLA evaluates the string expression and then emits that expression to the output .ASM file. See the HLA documentation for more details and restrictions on the #asm..#endasm block and the #emit directive.

4.12.8 The Conditional Compilation Statements (#if)

The conditional compilation statements in HLA use the following syntax:

```
#if( constant_boolean_expression )

    << Statements to compile if the >>
    << expression above is true.    >>

#elseif( constant_boolean_expression )

    << Statements to compile if the >>
    << expression immediately above >>
    << is true and the first expres->>
    << sion above is false.         >>

#else

    << Statements to compile if both    >>
    << the expressions above are false. >>

#endif
```

The #ELSEIF and #ELSE clauses are optional. As you would expect, there may be more than one #ELSEIF clause in the same conditional if sequence.

Unlike some other assemblers and high level languages, HLA's conditional compilation directives are legal anywhere whitespace is legal. You could even embed them in the middle of an instruction! While directly embedding these directives in an instruction isn't recommended (because it would make your code very hard to read), it's nice to know that you can place these directives in a macro and then replace an instruction operand with a macro invocation.

An important thing to note about this directive is that the constant expression in the #IF and #ELSEIF clauses must be of type boolean or HLA will emit an error. Any legal constant expression that produces a boolean result is legal here.

Keep in mind that conditional compilation directives are executed at compile-time, not at run-time. You would not use these directives to (attempt to) make decisions while your program is actually running.

5 The 80x86 Instruction Set in HLA

One of the most obvious differences between HLA and standard 80x86 assembly language is the syntax for the machine instructions. The two primary differences are the fact that HLA uses a functional notation for machine instructions and HLA arranges the operands in a (source, dest) format rather than the (dest, source) format used by Intel.

5.1 Zero Operand Instructions (Null Operand Instructions)

The following instructions do not require any operands. There are two sytactically allowable forms for each instruction:

```
instr;
instr();
```

The zero-operand instruction mnemonics are

aaa, aad, aam, aas, cbw, cdq, clc, cld, cli, cmc, cmpsb, cmpsd, cmpsw, cpuid, cwd, cwde, daa, das, insb, insd, insw, into, iret, iretd, lahf, leave, lodsb, lodsd, lodsw, movsb, movsd, movsw, nop, outsb, outsd, outsw, popad, popa, popf, popfd, pusha, pushad, pushf, pushfd, rdtsc, rep.insb, rep.insd, rep.insw, rep.movsb, rep.movsd, rep.movsw, rep.outsb, rep.outsd, rep.outsw, rep.stosb, rep.stosd, rep.stosw, repe.cmpsb, repe.cmpsd, repe.cmpsw, repe.scasb, repe.scasd, repe.scasw, repne.cmpsb, repne.cmpsd, repne.cmpsw, repne.scasb, repne.scasd, repne.scasw, sahf, scasb, scasd, scasw, stc, std, sti, stosb, stosd, stosw, wait, xlat

5.2 General Arithmetic and Logical Instructions

These instructions include adc, add, and, mov, or, sbb, sub, test, and xor. They all take the same basic form (substitute the appropriate mnemonic for "adc" in the syntax examples below):

Generic Form:

```
adc( source, dest );
```

Specific forms allowed:

```
adc( Reg8, Reg8 )
adc( Reg16, Reg16 )
adc( Reg32, Reg32 )

adc( const, Reg8 )
adc( const, Reg16 )
adc( const, Reg32 )

adc( const, mem )

adc( Reg8, mem )
adc( Reg16, mem )
adc( Reg32, mem )

adc( mem, Reg8 )
adc( mem, Reg16 )
adc( mem, Reg32 )

adc( Reg8, AnonMem )
adc( Reg16, AnonMem )
adc( Reg32, AnonMem )

adc( AnonMem, Reg8 )
adc( AnonMem, Reg16 )
adc( AnonMem, Reg32 )
```

Note: for the form "adc(const, mem)", if the specified memory location does not have a size or type associated with it, you must explicitly specify the size of the memory operand, e.g., "adc(5,(type byte [eax]));"

5.3 The XCHG Instruction

The xchg instruction allows the following syntactical forms:

Generic Form:

```
xchg( source, dest );
```

Specific Forms:

```
xchg( Reg8, Reg8 )
xchg( Reg8, mem )
xchg( Reg8, AnonMem)
xchg( mem, Reg8 )
xchg( AnonMem, Reg8 )

xchg( Reg16, Reg16 )
xchg( Reg16, mem )
xchg( Reg16, AnonMem)
xchg( mem, Reg16 )
xchg( AnonMem, Reg16 )

xchg( Reg32, Reg32 )
xchg( Reg32, mem )
```

```
xchg( Reg32, AnonMem)
xchg( mem, Reg32 )
xchg( AnonMem, Reg32 )
```

5.4 The CMP Instruction

The "cmp" instruction uses the following general forms:

Generic:

```
cmp( LeftOperand, RightOperand );
```

Specific Forms:

```
cmp( Reg8, Reg8 );
cmp( Reg8, mem );
cmp( Reg8, AnonMem );
cmp( mem, Reg8 );
cmp( AnonMem, Reg8 );
cmp( Reg8, const );

cmp( Reg16, Reg16 );
cmp( Reg16, mem );
cmp( Reg16, AnonMem );
cmp( mem, Reg16 );
cmp( AnonMem, Reg16 );
cmp( Reg16, const );

cmp( Reg32, Reg32 );
cmp( Reg32, mem );
cmp( Reg32, AnonMem );
cmp( mem, Reg32 );
cmp( AnonMem, Reg32 );
cmp( Reg32, const );

cmp( mem, const );
```

Note that the CMP instruction's operands are ordered "dest, source" rather than the usual "source,dest" format (that is, the operands are in the same order as MASM expects them). This is to allow an intuitive use of the instruction mnemonic (that is, CMP normally reads as "compare dest to source."). We will avoid this confusion by simply referring to the operands as the "left operand" and the "right operand". Left vs. right signifies the placement of the operands around a comparison operator like "<=" (e.g., "left <= right").

For the "cmp(mem, const)" form, the memory operand must have a type or size associated with it. When using anonymous memory locations you must always coerce the type of the memory location, e.g., "cmp((type word [ebp-4]), 0);".

5.5 The Multiply Instructions

HLA supports several variations on the 80x86 "MUL" and IMUL instructions. The supported forms are:

Standard Syntax:

```
mul( reg8 )
mul( reg16)
mul( reg32 )
mul( mem )

mul( reg8, al )
mul( reg16, ax )
mul( reg32, eax )

mul( mem, al )
mul( mem, ax )
mul( mem, eax )

mul( AnonMem, ax )
mul( AnonMem, dx:ax )
mul( AnonMem, edx:eax )

imul( reg8 )
imul( reg16)
imul( reg32 )
imul( mem )

imul( reg8, al )
imul( reg16, ax )
imul( reg32, eax )

imul( mem, al )
imul( mem, ax )
imul( mem, eax )

imul( AnonMem, ax )
imul( AnonMem, dx:ax )
imul( AnonMem, edx:eax )

intmul( const, Reg16 )
intmul( const, Reg16, Reg16 )
intmul( const, mem, Reg16 )
intmul( const, AnonMem, Reg16 )

intmul( const, Reg32 )
intmul( const, Reg32, Reg32 )
intmul( const, mem, Reg32 )
intmul( const, AnonMem, Reg32 )

intmul( Reg16, Reg16 )
intmul( mem, Reg16 )
intmul( AnonMem, Reg16 )

intmul( Reg32, Reg32 )
intmul( mem, Reg32 )
intmul( AnonMem, Reg32 )
```

Extended Syntax:

```
mul( const, al )
mul( const, ax )
mul( const, eax )
```

```
imul( const, al )
imul( const, ax )
imul( const, eax )
```

The first, and probably most important, thing to note about HLA's multiply instructions is that HLA uses a different mnemonic for the extended-precision integer multiply versus the single-precision integer multiply (i.e., IMUL vs. INTMUL). Standard MASM syntax uses the same mnemonic for both instructions. There are two reasons for this change of syntax in HLA. First, there needed to be some way to differentiate the "mul(const, al)" and the "intmul(const, al)" instructions (likewise for the instructions involving AX and EAX). Second, the behavior of the INTMUL instruction is substantially different from the IMUL instruction, so it makes sense to use different mnemonics for these instructions.

The extended syntax instructions create a static data variable, initialized with the specified constant, and then specify the address of this variable as the source operand of the MUL or IMUL instruction.

5.6 The Divide Instructions

HLA support several variations on the 80x86 DIV and IDIV instructions. The supported forms are:

Generic Forms:

```
div( source );
div( source, dest );

mod( source );
mod( source, dest );

idiv( source );
idiv( source, dest );

imod( source );
imod( source, dest );
```

Specific Forms:

```
div( reg8 )
div( reg16)
div( reg32 )
div( mem )

div( reg8, ax )
div( reg16, dx:ax)
div( reg32, edx:eax )

div( mem, ax )
div( mem, dx:ax)
div( mem, edx:eax )

div( AnonMem, ax )
div( AnonMem, dx:ax )
div( AnonMem, edx:eax )

mod( reg8 )
mod( reg16)
mod( reg32 )
mod( mem )
```

```
mod( reg8, ax )
mod( reg16, dx:ax)
mod( reg32, edx:eax )

mod( mem, ax )
mod( mem, dx:ax)
mod( mem, edx:eax )

mod( AnonMem, ax )
mod( AnonMem, dx:ax )
mod( AnonMem, edx:eax )

idiv( reg8 )
idiv( reg16)
idiv( reg32 )
idiv( mem )

idiv( reg8, ax )
idiv( reg16, dx:ax)
idiv( reg32, edx:eax )

idiv( mem, ax )
idiv( mem, dx:ax)
idiv( mem, edx:eax )

idiv( AnonMem, ax )
idiv( AnonMem, dx:ax )
idiv( AnonMem, edx:eax )

imod( reg8 )
imod( reg16)
imod( reg32 )
imod( mem )

imod( reg8, ax )
imod( reg16, dx:ax)
imod( reg32, edx:eax )

imod( mem, ax )
imod( mem, dx:ax)
imod( mem, edx:eax )

imod( AnonMem, ax )
imod( AnonMem, dx:ax )
imod( AnonMem, edx:eax )
```

Extended Syntax:

```
div( const, ax )
div( const, dx:ax )
div( const, edx:eax )

mod( const, ax )
mod( const, dx:ax )
mod( const, edx:eax )
```

```
idiv( const, ax )
idiv( const, dx:ax )
idiv( const, edx:eax )

imod( const, ax )
imod( const, dx:ax )
imod( const, edx:eax )
```

The destination operand is always implied by the 80x86 "div" and "idiv" instructions (AX, DX:AX, or EDX:EAX). HLA allows the specification of the destination operand in order to make your programs easier to read (although the use of the destination operand is optional).

The HLA divide instructions support an extended syntax that allows you to specify a constant as the divisor (source operand). HLA allocates storage in the static data segment and initializes the storage with the specified constant, and then divides the accumulator by this newly specified memory location.

5.7 Single Operand Arithmetic and Logical Instructions

These instructions include dec, inc, neg, and not. They take the following general forms (substituting the specific mnemonic as appropriate):

Generic Form:

```
dec( dest );
```

```
Specific forms allowed:
```

```
dec( Reg8 );
dec( Reg16 );
dec( Reg32 );
dec( mem );
```

Note: if mem is an untyped or unsized memory location (i.e., an anonymous memory location), you must explicitly provide a size; e.g., "dec((type word [edi]));"

5.8 Shift and Rotate Instructions

These instructions include RCL, RCR, ROL, ROR, SAL, SAR, SHL, and SHR. These instructions support the following generic syntax, making the appropriate mnemonic substitution.

Generic Form:

```
shl( count, dest );
```

Specific Forms:

```
shl( const, Reg8 );
shl( const, Reg16 );
shl( const, Reg32 );
```

```
shl( const, mem );

shl( cl, Reg8 );
shl( cl, Reg16 );
shl( cl, Reg32 );

shl( cl, mem );
```

The "const" operand is an unsigned integer constant between zero and the maximum number of bits in the destination operand. The forms with a memory operand must have a type or size associated with the operand; e.g., when using anonymous memory locations, you must coerce the type,

shl(2, (type dword [esi]));

5.9 The Double Precision Shift Instructions

These instruction use the following general form (you can substitute SHRD for SHLD below):

Generic Form:

```
shld( count, source, dest )
```

Specific Forms:

```
shld( const, Reg16, Reg16 )
shld( const, Reg16, mem )
shld( const, Reg16, AnonMem )

shld( cl, Reg16, Reg16 )
shld( cl, Reg16, mem )
shld( cl, Reg16, AnonMem )

shld( const, Reg32, Reg32 )
shld( const, Reg32, mem )
shld( const, Reg32, AnonMem )

shld( cl, Reg32, Reg32 )
shld( cl, Reg32, mem )
shld( cl, Reg32, AnonMem )
```

5.10 The Lea Instruction

These instructions use the following syntax:

```
lea( Reg32, memory )
lea( Reg32, AnonMem )
lea( Reg32, ProcID )
lea( Reg32, LabelID )
```

Extended Syntax:

```
lea( memory, Reg32 )
lea( AnonMem, Reg32 )
lea( ProcID, Reg32 )
lea( LabelID, Reg32 )
lea( StringConstant, Reg32 )
lea( const ConstExpr, Reg32 )
```

The "lea" instruction loads the specified 32-bit register with the address of the specified memory operand, procedure, or statement label. Note in the extended syntax you can reverse the order of the operands. Since exactly one operand must be a register, there is no ambiguity between the two forms (this syntax was added to satisfy those who complained about the (reg,memory) syntax). Of course, good programming style suggests that you use only one form (either reg,memory or memory, reg) within your programs.

Note: HLA does not support an LEA instruction that loads a 16-bit address into a 16-bit register. That form of the LEA instruction is not very useful in 32-bit programs running on 32-bit operating systems.

5.11 The Sign and Zero Extension Instructions

The HLA MOVSX and MOVZX instructions use the following syntax:

Generic Forms:

```
movsx( source, dest );
movzx( source, dest );
```

Specific Forms:

```
movsx( Reg8, Reg16 )
movsx( Reg8, Reg32 )
movsx( Reg16, Reg32 )
movsx( mem8, Reg16 )
movsx( mem8, Reg32 )
movsx( mem16, Reg32 )

movzx( Reg8, Reg16 )
movzx( Reg8, Reg32 )
movzx( Reg16, Reg32 )
movzx( mem8, Reg16 )
movzx( mem8, Reg32 )
movzx( mem16, Reg32 )
```

These instructions sign (MOVSX) or zero (MOVZX) extend their source operand into the destination operand.

5.12 The Push and Pop Instructions

These instructions take the following general forms:

```
pop( reg16 );
pop( reg32 );
pop( mem );

push( Reg16 )
push( Reg32 )
```

```
push( memory )

pushw( Reg16 )
pushw( memory )
pushw( AnonMem )
pushw( Const )

pushd( Reg32 )
pushd( memory )
pushd( AnonMem )
pushd( Const )
```

These instructions push or pop their specified operand.

5.13 Procedure Calls

Given a procedure or a DWORD variable (containing the address of a procedure) named "MyProc" you can call this procedure as follows:

```
call( MyProc );
```

HLA actually supports several other syntaxes for calling procedures, including a syntax that will automatically push parameters on the stack for you. See the HLA Reference Manual for more details.

5.14 The Ret Instruction

The RET( ) statement allows two syntactical forms:

```
ret( );
ret( integer_constant_expression );
```

The first form emits a simple 80x86 RET instruction, the second form emits the 80x86 RET instruction with the specified numeric constant expression value (used to remove parameters from the stack).

5.15 The Jmp Instructions

The HLA "jmp" instruction supports the following syntax:

```
jmp    Label;
jmp    ProcedureName;
jmp( dwordMemPtr );
jmp( anonMemPtr );
jmp( reg32 );
```

"Label" represents a statement label in the current procedure. (You are not allowed to jump to labels in other procedures in the current version of HLA. This restriction may be relaxed somewhat in future versions.) A statement label is a unique (within the current procedure) identifier with a colon after the identifier, e.g.,

```
InfiniteLoop:
    << Code inside the infinite loop>>
    jmp InfiniteLoop;
```

Jumping to a procedure transfers control to the first instruction in the specified procedure. You are responsible for explicitly pushing any parameters and the return address for that procedure.

These instructions all return the empty string as their "returns" value.

5.16 The Conditional Jump Instructions

These instructions include JA, JAE, JB, JBE, JC, JE, JG, JGE, JL, JLE, JO, JP, JPE, JPO, JS, JZ, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JCXZ, JECXZ, LOOP, LOOPE, LOOPZ, LOOPNE, and LOOPNZ. They all take the following generic form (substituting the appropriate instruction for "JA").

```
ja      LocalLabel;
```

"LocalLabel" must be a statement label defined in the current procedure.

Note: due to the nature of the HLA compilation process, you should avoid the use of the JCXZ, JECXZ, LOOP, LOOPE, LOOPZ, LOOPNE, and LOOPNZ instructions. Unlike the other conditional jump instructions, these instructions have a very limited +/- 128 range. Unfortunately, HLA cannot detect if the branch is out of range (this task is handled by MASM), so if a range error occurs, HLA cannot warn you about this. The MASM assembly will fail, but the result will be hard to decipher. Fortunately, these instructions are easily, and usually more efficiently, implemented using other 80x86 instructions so this should not prove to be a problem.

5.17 The Conditional Set Instructions

These instructions include: SETA, SETAE, SETB, SETBE, SETC, SETE, SETG, SETGE, SETL, SETLE, SETO, SETP, SETPE, SETPO, SETS, SETZ, SETNA, SETNAE, SETNB, SETNBE, SETNC, SETNE, SETNG, SETNGE, SETNL, SETNLE, SETNO, SETNP, SETNS, and SETNZ. They take the following generic forms (substituting the appropriate mnemonic for seta):

```
seta( Reg8 );
seta( mem );
seta( AnonMem );
```

5.18 The Conditional Move Instructions

These instructions include CMOVA, CMOVAE, CMOVB, CMOVBE, CMOVC, CMOVE, CMOVG, CMOVGE, CMOVL, CMOVLE, CMOVO, CMOVP, CMOVPE, CMOVPO, CMOVS, CMOVZ, CMOVNA, CMOVNAE, CMOVNB, CMOVNBE, CMOVNC, CMOVNE, CMOVNG, CMOVNGE, CMOVNL, CMOVNLE, CMOVNO, CMOVNP, CMOVNS, and CMOVNZ. They use the following general syntax:

```
CMOVcc( src, dest );
```

Allowable operands:

```
CMOVcc( reg16, reg16 );
CMOVcc( reg32, reg32 );
CMOVcc( mem16, reg16 );
CMOVcc( mem32, reg32 );
```

These instructions move the data if the specified condition is true (specified by the *cc* condition). If the condition is false, these instructions behave like a no-operation.

5.19 The Input and Output Instructions

The "in" and "out" instructions use the following syntax:

```
in( port, al )
in( port, ax )
in( port, eax )

in( dx, al )
in( dx, ax )
in( dx, eax )

out( al, port )
out( ax, port )
out( eax, port )

out( al, dx )
out( ax, dx )
out( eax, dx )
```

The "port" parameter must be an unsigned integer constant in the range 0..255. Note that these instructions may be priviledged instructions when running under Win32. Their use may generate a fault in certain instances or when accessing certain ports.

5.20 The Interrupt Instruction

This instruction uses the syntax "int(constant)" where the constant operand is an unsigned integer value in the range 0..255.

This instruction returns the empty string as its "returns" value.

See Chapter Six in "Art of Assembly" (DOS version) for a further discussion of this instruction. Note, however, that one generally does not use "int" under Win32 to make OS or BIOS calls. The "int $80" instruction is what you'd normally use to make very low-level Linux calls.

5.21 Bound Instruction

This instruction takes the following forms:

```
bound( Reg16, mem )
bound( Reg16, AnonMem )

bound( Reg32, mem )
bound( Reg32, AnonMem )
```

Extended Syntax Form:

```
bound( Reg16, constL, constH )
bound( Reg32, ConstL, ConstH )
```

The extended syntax forms emit the two constants to the static data segment and substitute the address of the first constant (ConstL) as their memory operand.

5.22 The Enter Instruction

The ENTER instruction uses the syntax: "enter(const, const);". The first constant operand is the number of bytes of local variables in a procedure, the second constant operand is the lex level of the procedure. As a general rule, you should not use this instruction (and the corresponding LEAVE) instructions. HLA procedures automatically construct the display and activation record for you (more efficiently than when using ENTER). See the HLA Reference Manual for more details on building procedure activation records.

5.23 CMPXCHG Instruction

This instruction uses the following syntax:

Generic Form:

```
cmpxchg( reg/mem, reg )
```

Specific Forms:

```
cmpxchg( Reg8, Reg8 )
cmpxchg( Reg8, Memory )
cmpxchg( Reg8, AnonMem )

cmpxchg( Reg16, Reg16 )
cmpxchg( Reg16, Memory )
cmpxchg( Reg16, AnonMem )

cmpxchg( Reg32, Reg32 )
cmpxchg( Reg32, Memory )
cmpxchg( Reg32, AnonMem )
```

Note: HLA does not currently support the Pentium cmpxchg8b instruction, although you can easily create a macro to implement this instruction. You can also use #ASM..#ENDASM or #EMIT to implement the CMPXCHG8B instruction.

5.24 The XADD Instruction

The XADD instruction uses the following syntax:

Generic Form:

```
xadd( source, dest );
```

Specific Forms:

```
xadd( Reg8, Reg8 )
xadd( mem, Reg8 )
xadd( AnonMem, Reg8 )

xadd( Reg16, Reg16 )
xadd( mem, Reg16 )
xadd( AnonMem, Reg16 )

xadd( Reg32, Reg32 )
xadd( mem, Reg32 )
xadd( AnonMem, Reg32 )
```

5.25 BSF and BSR Instructions

The bit scan instructions use the following syntax (substitute BSR for BSF as appropriate):

Generic Form:

```
bsr( source, dest );
```

Specific Forms Allowed:

```
bsf( Reg16, Reg16 );
bsf( mem, Reg16 );
bsf( AnonMem, Reg16 );

bsf( Reg32, Reg32 );
bsf( mem, Reg32 );
bsf( AnonMem, Reg32 );
```

5.26 The BSWAP Instruction

This instruction takes the form "bswap(reg32)". It converts between little endian and big endian data formats in the specified 32-bit register.

5.27 Bit Test Instructions

This group of instructions includes BT, BTC, BTR, and BTS. They allow the following generic forms:

Generic Form:

```
bt( BitNumber, Dest );
```

Specific Forms:

```
bt( const, Reg16 );
bt( const, Reg32 );

bt( const, mem );

bt( Reg16, Reg16 );
bt( Reg16, mem );
bt( Reg16, AnonMem );

bt( Reg32, Reg32 );
bt( Reg32, mem );
bt( Reg32, AnonMem );
```

Substitute the BTC, BTR, or BTS mnemonic for BT in the examples above for these other instructions.

5.28 Floating Point Instructions

HLA supports the following FPU instructions.

```
fld( FPreg );
fst( FPreg );

fld( FPmem );           // Returns operand.
fst( FPmem );           // 32 and 64-bits only!  Returns operand.
fstp( FPmem );          // Returns operand.

fxch( FPreg );

fild( FPmem );          // Returns operand.
fist( FPmem );          // 32 and 64-bits only!  Returns operand.
fistp( FPmem );         // Returns operand.

fbld( FPmem );          // Returns operand.
fbstp( FPmem );         // Returns operand.

fadd( );
fadd( FPreg, st0 );
fadd( st0, FPreg );
fadd( FPmem );          // Returns operand.
fadd( FPconst );        // Returns operand.

faddp( );
faddp( st0, FPreg );
```

```
fmul( );
fmul( FPreg, st0 );
fmul( st0, FPreg );
fmul( FPmem );          // Returns operand.
fmul( FPconst );        // Returns operand.

fmulp( );
fmulp( st0, FPreg );

fsub( );
fsub( FPreg, st0 );
fsub( st0, FPreg );
fsub( FPmem );          // Returns operand.
fsub( FPconst );        // Returns operand.

fsubp( );
fsubp( st0, FPreg );

fsubr( );
fsubr( FPreg, st0 );
fsubr( st0, FPreg );
fsubr( FPmem );         // Returns operand.
fsubr( FPconst );       // Returns operand.

fsubrp( );
fsubrp( st0, FPreg );

fdiv( );
fdiv( FPreg, st0 );
fdiv( st0, FPreg );
fdiv( FPmem );          // Returns operand.
fdiv( FPconst );        // Returns operand.

fdivp( );
fdivp( st0, FPreg );

fdivr( );
fdivr( FPreg, st0 );
fdivr( st0, FPreg );
fdivr( FPmem );         // Returns operand.
fdivr( FPconst );       // Returns operand.

fdivrp( );
fdivrp( st0, FPreg );

fiadd( mem16 );         // Returns operand.
fiadd( mem32 );         // Returns operand.
fiadd( const );         // Returns operand.

fimul( mem16 );         // Returns operand.
fimul( mem32 );         // Returns operand.
fimul( const );         // Returns operand.

fidiv( mem16 );         // Returns operand.
fidiv( mem32 );         // Returns operand.
fidiv( mem32 );         // Returns operand.
fidiv( const );         // Returns operand.

fidivr( mem16 );        // Returns operand.
fidivr( mem32 );        // Returns operand.
fidivr( const );        // Returns operand.
```

```
fcom( );
fcom( FPreg );
fccom( FPmem );          // Returns operand.

fcomp( );
fcomp( FPreg );
fcomp( FPmem );          // Returns operand.

fucom( );
fucom( FPreg );

fucomp( );
fucomp( FPreg );

fcompp();
fucompp();


ficom( mem16 );          // Returns operand.
ficom( mem32 );          // Returns operand.
ficom( const );          // Returns operand.

ficomp( mem16 );         // Returns operand.
ficomp( mem32 );         // Returns operand.
ficomp( const );         // Returns operand.

fsqrt();             // The following all return "st0"
fscale();
fprem();
fprem1();
frndint();
fxtract();
fabs();
fchs();
ftst();
fxam();
fldz();
fld1();
fldpi();
fldl2t();
fldl2e();
fldlg2();
fldln2();
f2xm1();
fsin();
fcos();
fsincos();
fptan();
fpatan();
fyl2x();
fyl2xp1();

finit();             // Returns ""
fwait();
fclex();
fincstp();
fdecstp();
fnop();
ffree( FPreg );
```

```
fldcw( mem );
fstcw( mem );
fstsw( mem );
```

See the chapter on real arithmetic in "The Art of Assembly Language Programming" for details on these instructions. Note that HLA does not support the entire FPU instruction set. If you absolutely need the few remaining instructions, use the #ASM..#ENDASM or #EMIT directives to generate them.

5.29 Additional Floating Point Instructions for Pentium Pro and Later Processors

The FCMOVcc instructions (cc= a, ae, b, be, na, nae, nb, nbe, e, ne, u, nu) use the following basic syntax:

```
FCMOVcc( stn, st0);  // n=0..7
```

They move the specified floating point register to ST0 if the specified condition is true.

The FCOMI and FCOMIP instructions use the following syntax:

```
fcomi( st0, stn );
fcomip( st0, stn );
```

These instructions behave like their (syntactical equivalent) FCOM and FCOMP brethren except they store the status in the EFLAGs register directly rather than in the floating point status register.

5.30 MMX Instructions

HLA supports the following MMX instructions found on the Pentium and later processors (note that some instructions are only available on Pentium III and later processors; see the Intel reference manuals for details):

HLA uses the symbols mm0, mm1, ..., mm7 for the MMX register set.

The following MMX instructions all use the same syntax. The syntax is

```
mmxInstr( mmxReg, mmxReg );
mmxInstr( mem64, mmxReg );
```

mmxInstrs:

```
paddb
paddw
paddd
paddsb
paddsw
paddusb
paddusw

psubb
psubw
psubd
psubsb
psubsw
psubusb
psubusw
```

```
pmulhuw
pmulhw
pmullw
pmaddwd

pavgb
pavgw

pcmpeqb
pcmpeqw
pcmpeqd
pcmpgtb
pcmpgtw
pcmpgtd

packsswb
packuswb
packssdw

punpcklbw
punpcklwd
punpckldq
punpckhbw
punpckhwd
punpckhdq

pand
pandn
por
pxor

psllw
pslld
psllq
psrlw
psrld
psrlq
psraw
psrad

pmaxsw
pmaxub

pminsw
pminub

psadbw
```

The following MMX instructions require a special syntax. The syntax is listed for each instruction.

```
pextrw(  constant, mmxReg, Reg32 );
pinsrw( constant, Reg32, mmxReg );
pmovmskb( mmxReg, Reg32 );
pshufw( constant, mmxReg, mmxReg );
pshufw( constant, mem64, mmxReg );

movd( mem32, mmxReg );
movd( mmxReg, mem32 );
movq( mem64, mmxReg );
movq( mmxReg, mem64 );
```

```
emms();
```

Please see the appropriate Intel documentation or "The Art of Assembly Language" for a discussion of the behavior of these instructions.

6 Memory Addressing Modes in HLA

HLA supports all the 32-bit addressing modes of the Intel 80x86 instruction set[2]. A memory address on the 80x86 may consist of one to three different components: a displacement (also called an offset), a base pointer, and a scaled index value. The following are the legal combinations of these components:

```
displacement
basePointer
displacement + basePointer
displacement + scaledIndex
basePointer + scaledIndex
displacement + basePointer + scaledIndex
```

Note that a scaled index value cannot exist by itself.

HLA's syntax for memory addressing modes takes the following forms:

```
staticVarName


staticVarName [ constant ]

staticVarName[ breg32 ]
staticVarName[ ireg32 ]
staticVarName[ ireg32*index ]


staticVarName[ breg32 + ireg32 ]
staticVarName[ breg32 + ireg32*index ]

staticVarName[ breg32 + constant ]
staticVarName[ ireg32 + constant ]

staticVarName[ ireg32*index + constant ]

staticVarName[ breg32 + ireg32 + constant ]
staticVarName[ breg32 + ireg32*index + constant ]

staticVarName[ breg32 - constant ]
staticVarName[ ireg32 - constant ]
staticVarName[ ireg32*index - constant ]

staticVarName[ breg32 + ireg32 - constant ]
staticVarName[ breg32 + ireg32*index - constant ]


[ breg32 ]
```

2. It does not support the 16-bit addressing modes since these are not very useful under Win32.

```
[ breg32 + ireg32 ]
[ breg32 + ireg32*index ]

[ breg32 + constant ]

[ breg32 + ireg32 + constant ]
[ breg32 + ireg32*index + constant ]

[ breg32 - constant ]

[ breg32 + ireg32 - constant ]
[ breg32 + ireg32*index - constant ]
```

"staticVarName" denotes any static variable currently in scope (local or global).

"basereg" denotes any general purpose 32-bit register.

"$breg_{32}$" denotes a base register and can be any general purpose 32-bit register.

"$ireg_{32}$" denotes an index register and may also be any general purpose register, even the same register as the base register in the address expression.

"index" denotes one of the four constants "1", "2", "4", or "8". In those address expression that have an index register without an index constant, "*1" is the default index.

Those memory addressing modes that do not have a variable name preceding them are known as "anonymous memory locations." Anonymous memory locations do not have a data type associated with them and in many instances you must use the type coercion operator in order to keep HLA happy.

Those memory addressing modes that do have a variable name attached to them inherit the base type of the variable. Read the next section for more details on data typing in HLA.

HLA allows another way to specify addition of the various addressing mode components in an address expression – by putting the components in separate brackets and concatenating them together. The following examples demonstrate the standard syntax and the alternate syntax:

```
[ebx+2]              [ebx][2]
[ebx+ecx*4+8]        [ebx][ecx][8]
lbl[ebp-2]           lbl[ebp][-2]
```

The reason for allowing the extended syntax is because you might want to construct these addressing modes inside a macro from the individual pieces and it's much easier to concatenate two operands already surrounded by brackets than it is to pick the expressions apart and construct the standard addressing mode.

7 Type Coercion in HLA

While an assembly language can never really be a strongly typed language, HLA is much more strongly typed than most other assembly languages.

Strong typing in an assembly language can be very frustrating. Therefore, HLA makes certain concessions to prevent the type system from interfering with the typical assembly language programmer. Within an 80x86 machine instruction, the only checking that takes place is a verification that the sizes of the operands are compatible.

Despite HLA playing fast and loose with machine instructions, there are many times when you will need to coerce the type of some operand. HLA uses the following syntax to coerce the type of a memory location or register operand:

```
(type typeID  memOrRegOperand)
```

There are two instances where type coercion is especially important: (1) when you need to assign a type other than byte, word, or dword to a register[3]; (2) when you need to assign an anonymous memory location a type.

3. Probably the most common case is treating a register as a signed integer in one of HLA's high level language statements. See the section on HLA High Level Language statements for more details.