2 The Quick Guide to HLA

2.1 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.

2.2 Running HLA

HLA is a command line tool that you run from the Win32, Mac OSX, Linux, or FreeBSD 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" (under Windows) or "ls" and "mv" (under *NIX). 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 base name and an .obj/.o extension). Finally, HLA runs the linker to combine all the object files together. The ultimate result, assuming there were no errors along the way, is an executable file .

HLA supports the following command line parameters:

```
options:
 -@          Do not generate linker response file.
 -@@         Always generate a linker response file.
 -thread     Enable thread-safe code generation and linkage.
 -axxxxx     Pass xxxxx as command line parameter to assembler.
 -dxx        Define VAL symbol xx to have type BOOLEAN and value TRUE.
 -dxx=yy     Define VAL symbol xx to have type STRING and value "yy".
 -e:name     Executable output filename (appends ".exe" under Windows).
 -x:name     Executable output filename (does not append ".exe").
 -b:name     Binary object file output name (only when using HLABE).
 -i:path     Specifies path to HLA include file directory.
 -lib:path   Specifies path to the HLALIB.LIB file.
```

```
-license     Displays copyright and license info for the HLA system.
-lxxxxx      Pass xxxxx as command line parameter to linker.
-m           Create a map file during link
-p:path      Specifies path to hold temporary working files.
-r:name      <name> is a text file containing cmd line options.
-obj:path    Specifies path to place object files.
-main:name   Use 'name' as the name of the HLA main program.
-source      Compile to human readable source file format.
-s           Compile to .ASM files only.
-c           Compile and assemble to object files only.
-fasm        Use FASM as back-end assembler (applies to -s and -c)
-gas         Use GAS as back-end (Linux/BSD, applies to -s and -c)
-gasx        Use Gas as back-end (Mac OSX, --s and -c only)
-hla         Produce a pseudo-HLA source file as output (implies -s).
-hlabe       (Default) Produce object code using the HLA Back Engine.
-masm        Use MASM as back-end assembler (applies to -s and -c)
-nasm        Use NASM as back-end assembler (applies to -s and -c)
-tasm        Use TASM as back-end assembler (applies to -s and -c)
-sym         Dump symbol table after compile.
-win32       Generate code for Win32 OS.
-linux       Generate code for Linux OS.
-freebsd     Generate code for FreeBSD OS.
-macos       Generate code for Mac OSX.
-test        Send diagnostic info to stdout rather than stderr (This
              option is intended for HLA test/debug purposes).
-v           Verbose compile.
-level=h     High-level assembly language
-level=m     Medium-level assembly language
-level=l     Low-level assembly language
-level=v     Machine-level assembly language (very low level).
-w           Compile as windows app (default is console app).
-?           Display this help message.
```

Please see the appropriate chapter in the HLA Reference Manual chapter *Using the HLA Command-Line Compiler* 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).

2.3 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 or *NIX (Mac OSX, Linux, and FreeBSD) source files with a line feed terminating each line. 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.

2.3.1 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.

2.3.2 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.

2.3.3 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 chapter *HLA Language Elements* 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 respect to case) both represent the HLA "mov" reserved word.

2.3.4 External Symbols and Assembler Reserved Words

HLA v2.0 produces an option to produce an assembly language file during compilation and can invoke an assembler such as MASM, FASM, NASM, or Gas to complete the compilation process. HLA automatically translates normal identifiers you declare in your program to benign 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 the back-end assembler's reserved words, please see the documentation for the assembler you are using to process HLA’s output (i.e., MASM, NASM, FASM, or Gas).

2.3.5 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 assemblers and linkers that process HLA output 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.

2.3.6 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 back-end assembler (if you are using one). 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.

2.4 Data Types in HLA

2.4.1 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 chapter *HLA Data Types*. This document will make use of the following types:

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

These are the typical types assembly language programmers use.

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. TBYTE objects consume ten bytes (80 bits). 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.

2.4.2 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).

2.4.3 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 indices 0..15).

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

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

2.4.4 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:
```

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."

```
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.,

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

2.5 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.

2.5.1 Numeric Constants

HLA lets you specify several different types of numeric constants.

2.5.1.1 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.

2.5.1.2 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.

2.5.1.3 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.

2.5.1.4 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.

2.5.1.5 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.

2.5.1.6 Character Constants

Character literals generally consist of a single (graphic) character surrounded by apostrophes. To represent the apostrophe character, you use four apostrophes, 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.

2.5.1.7 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.

2.5.1.8 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 (**storage**) variable, procedure, method, or iterator), HLA will compute the run-time offset of that variable. Pointer constants may not be used in arbitrary constant expressions. You may only use pointer constants in expressions used to initialize static or **readonly** variables or as constant expressions in 80x86 instructions.

2.5.1.9 Structured Constants

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

2.6 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 chapter *HLA Constants* for more details.

2.7 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, storage, readonly, procedure, iterator,** and **method** 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;
```

2.8 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 many other features this document won't describe. For more details, please see the HLA Reference Manual chapter on *HLA Procedures*.

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;
```

2.8.1 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 legal:

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

     //...

 end TwoConsts;
```

2.8.2 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 identifier followed by a colon and a type definition. The following paragraphs demonstrate some of the legal forms of type definitions. See the HLA Reference Manual chapter on *HLA Program Structure* for more examples.

```
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;
```

2.8.3 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 fixed 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 chapter on *HLA Program Structure* for more details.

2.8.4 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 chapter on *HLA Program Structure* 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
```

2.8.4.1 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.

2.8.4.2 The EXTERNAL Option

The **external** option gives you the ability to reference 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 variable 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 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.

2.8.5 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 chapter *The HLA Compile-Time Language* to learn all about HLA's powerful macro facilities. This section will describe HLA's limited "Standard Macro" facility that 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 legal:

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

This expands to:

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

2.9 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 );
```

2.10 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.

2.11 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.

2.11.1 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

2.11.2 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:

Generic Form:

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

```
mov( source, dest );
sbb( source, dest );
sub( source, dest );
test( source, dest );
xor( source, dest );
```

2.11.3 The XCHG Instruction

The xchg instruction allows the following syntactical forms:

Generic Form:

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

2.11.4 The CMP Instruction

The "cmp" instruction uses the following general forms:

Generic:

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

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").

2.11.5 The Multiply Instructions

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

Standard Syntax:

```
mul( src )
imul( src )


intmul( const, Reg )
intmul( const, Reg, Reg )
intmul( Reg, Reg )
intmul( mem, Reg )
```

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**).

Note that the forms listed above correspond to the standard mul and imul instructions most assemblers provide. HLA actually provides several additional forms, please see the HLA documentation on "The 80x86 Instruction Set in HLA" for more details.

2.11.6 The Divide Instructions

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

Generic Forms:

```
div( source );
idiv( source );
```

Note that the forms listed above correspond to the standard **div** and **idiv** instructions most assemblers provide. HLA actually provides several additional forms; please see the HLA Reference manual chapter on **The 80x86 Instruction Set in HLA** for more details.

2.11.7 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 );
inc( dest );
neg( dest );
not( dest );
```

2.11.8 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 );
shr( count, dest );
sar( count, dest );
sal( count, dest );
rcl( count, dest );
rcr( count, dest );
rol( count, dest );
ror( count, dest );
```

2.11.9 The Double Precision Shift Instructions

These instruction use the following general form:

Generic Form:

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

```
shrd( count, source, dest )
```

2.11.10 The Lea Instruction

These instructions use the following syntax:

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

lea( Reg32, LabelID )
```

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 useful in 32-bit programs running on 32-bit operating systems.

2.11.11 The Sign and Zero Extension Instructions

The HLA **movsx** and **movzx** instructions use the following syntax:

Generic Forms:

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

2.11.12 The Push and Pop Instructions

These instructions take the following general forms:

```
pop( reg );
pop( mem );
pushw( Reg16 )
pushw( memory )
pushw( Const )

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

These instructions push or pop their specified operand.

2.11.13 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 chapter on *HLA Procedures* for more details.

2.11.14 The Ret Instruction

The **ret** statement allows two syntactical forms:

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

2.11.15 The Jmp Instructions

The HLA **jmp** instruction supports the following syntax:

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

2.11.16 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;
```

2.11.17 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 );
```

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(reg_{16}, reg_{16});

CMOVcc(reg_{32}, reg_{32});

CMOVcc(mem_{16}, reg_{16});

CMOVcc(mem_{32}, reg_{32});

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.

2.11.18 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 privileged instructions when running under 32-bit operating systems. Their use may generate a fault in certain instances or when accessing certain ports.

2.11.19 The Interrupt Instruction

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

2.11.20 Bound Instruction

This instruction takes the following form:

```
bound( Reg16/32, mem )
```

2.11.21 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 rule, you should not use this instruction and the corresponding **leave** instruction. HLA procedures automatically construct the display and activation record for you (more efficiently than when using **enter**). See the HLA Reference Manual chapter on *HLA Procedures* for more details on building procedure activation records.

2.11.22 CMPXCHG Instruction

This instruction uses the following syntax:

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

2.11.23 The XADD Instruction

The XADD instruction uses the following syntax:

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

2.11.24 BSF and BSR Instructions

The bit scan instructions use the following syntax:

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

2.11.25 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.

2.11.26 Bit Test Instructions

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

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

2.11.27 Floating Point Instructions

See the HLA Reference Manual chapter *The 80x86 Instruction Set in HLA* for a complete list of the floating-point instructions and their syntax.

2.11.28 MMX and SSE Instructions

See the HLA Reference Manual chapter *The 80x86 Instruction Set in HLA* for a complete list of the MMX and SSE instructions and their syntax.

2.12 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:

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

```
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 ]

[ 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.

2.13 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.