16 The HLA Memory Model and Memory Addressing Modes

This chapter describes how HLA views memory at run time and how individual instructions can access memory.

16.1 The HLA Memory Model

HLA uses a variant of the standard *a.out* memory model. The *a.out* memory model organizes data in an executable file into three distinct segments (or sections) and organizes run-time memory allocation into five distinct sections. Though it is possible to create fancier memory models, the *a.out* memory model is a time-proven tried-and-true memory model that supports the creation of almost any imaginable executable file.

During compilation, HLA organizes the object code it produces into one of four sections: a *code* (or *text*) section, a *readonly* data section, a static (initialized) *data* section, and a static (uninitialized) *bss* section. (*bss* is an old assembly language term that stands for Block Started by Symbol; the modern meaning of this term is any block of variables that aren't given a non-zero initial value when loaded into memory.) When HLA writes the object file to disk, it combines the *readonly* and *text/code* sections into a single section in the object file (the *text* section); therefore, there are only three sections of interest in the object code file. The object file might also contain other data such as symbol tables, string tables, and relocation tables, but such data is not present in the run-time code and is of no interest to us here.

When the operating system loads an HLA executable file into memory, it loads the *text*, *data*, and *bss* sections into their respective regions in memory before transferring control to the main program (which will be in the *text* section). In addition to these three sections that exist in the executable file's disk image, an HLA program generally references two other sections of memory at run time: the *stack* area and the *heap*. These two areas are created dynamically at run time by the operating system and the HLA run-time system.

The *text* and the *data* sections in memory correspond almost byte-for-byte with their respective sections in the executable disk file. Indeed, the only difference between the sections in the disk image and the sections in memory at run time is that the run-time image may have been *relocated* to a new address.

The *bss* section disk image doesn't contain any actual data. This is just a data structure in the disk image that tells the operating system how much storage to set aside when loading the executable into memory. The operating system will allocate the storage, fill it with zero bytes, and then adjust all the addresses in the *text* and *data* sections that reference the *bss* section. The size of the bss section is solely determined by the number of bytes of variables declared in the **storage** declaration sections of the HLA program.

Different operating systems arrange the text, data, and bss sections differently in memory; however, all of the data in one such section usually resides in one contiguous block of run-time memory. Multi-threaded applications can have multiple stacks, but the program generally starts with one stack section. The number of heap sections in memory depends entirely on how the operating system implements memory allocation.

You should realize that the HLA *text/code* section may contain data as well as machine instructions. Data you declare in an HLA **readonly** section and any necessary constants (such as string constants that HLA generates) are merged in with the machine instructions in the *text* section.

16.2 Memory Addressing Modes in HLA

HLA supports all the 32-bit addressing modes of the Intel 80x86 instruction set[1]. A memory address on the 80x86 may consist of one to three different components: a displacement (also called

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

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
```

The following addressing modes are legal, but are mainly useful only within an **lea** instruction:

```
scaledIndex
scaledIndex + displacement
```

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

staticVarName

staticVarName [constant]

staticVarName[$breg_{32}$]
staticVarName[$ireg_{32}$]
staticVarName[$ireg_{32}$*index]

staticVarName[$breg_{32}$ + $ireg_{32}$]
staticVarName[$breg_{32}$ + $ireg_{32}$*index]

staticVarName[$breg_{32}$ + constant]
staticVarName[$ireg_{32}$ + constant]

staticVarName[$ireg_{32}$*index + constant]

staticVarName[$breg_{32}$ + $ireg_{32}$ + constant]
staticVarName[$breg_{32}$ + $ireg_{32}$*index + constant]

staticVarName[$breg_{32}$ - constant]
staticVarName[$ireg_{32}$ - constant]
staticVarName[$ireg_{32}$*index - constant]

staticVarName[$breg_{32}$ + $ireg_{32}$ - constant]
staticVarName[$breg_{32}$ + $ireg_{32}$*index - constant]

localVarName

localVarName [constant]

localVarName[$ireg_{32}$]
localVarName[$ireg_{32}$*index]

```
localVarName[ ireg32 + constant ]
localVarName[ ireg32*index + constant ]

localVarName[ ireg32 - constant ]
localVarName[ ireg32*index - constant ]


basereg:globalVarName

basereg:globalVarName [ constant ]

basereg::globalVarName[ ireg32 ]
basereg::globalVarName[ ireg32*index ]

basereg::globalVarName[ ireg32 + constant ]
basereg::globalVarName[ ireg32*index + constant ]

basereg::globalVarName[ ireg32 - constant ]
basereg::globalVarName[ 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 ]
```

The following are legal, but are only useful within the **lea** instruction:

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

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

"localVarName" denotes a local, automatic, variable declared in the **var** section of the current procedure.

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

"globalVarname" denotes a non-local variable declared in the **var** section of some procedure other than the current procedure.

"$breg_{32}$" 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 except ESP, 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*4][8]
lbl[ebp-2]             lbl[ebp][-2]
[ ebx*8 + 5 ]          [ebx*8][5]
```

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.

In general, the extended syntax takes one of the following forms (braces surround optional items):

[constExpr] { *<<additional address items inside "[]">>* }

[$base_{32}$] { *<<additional address items inside "[]">>* }

[$index_{32}$*1] { *<<additional address items inside "[]">>* }
[$index_{32}$*2] { *<<additional address items inside "[]">>* }
[$index_{32}$*4] { *<<additional address items inside "[]">>* }
[$index_{32}$*8] { *<<additional address items inside "[]">>* }

[$base_{32}$+$index_{32}$] { *<<additional address items inside "[]">>* }
[$base_{32}$+$index_{32}$*1] { *<<additional address items inside "[]">>* }
[$base_{32}$+$index_{32}$*2] { *<<additional address items inside "[]">>* }
[$base_{32}$+$index_{32}$*4] { *<<additional address items inside "[]">>* }
[$base_{32}$+$index_{32}$*8] { *<<additional address items inside "[]">>* }

[$base_{32}$ + constExpr] { *<<additional address items inside "[]">>* }

[$index_{32}$*1 + constExpr] { *<<additional address items inside "[]">>* }
[$index_{32}$*2 + constExpr] { *<<additional address items inside "[]">>* }
[$index_{32}$*4 + constExpr] { *<<additional address items inside "[]">>* }
[$index_{32}$*8 + constExpr] { *<<additional address items inside "[]">>* }

[$base_{32}$+$index_{32}$ + constExpr] { *<<additional address items inside "[]">>* }
[$base_{32}$+$index_{32}$*1 + constExpr] { *<<additional address items inside "[]">>*
}
[$base_{32}$+$index_{32}$*2 + constExpr] { *<<additional address items inside "[]">>*
}
[$base_{32}$+$index_{32}$*4 + constExpr] { *<<additional address items inside "[]">>*
}
[$base_{32}$+$index_{32}$*8 + constExpr] { *<<additional address items inside "[]">>*
}

[base$_{32}$ - constExpr] { *<<additional address items inside "[]">>* }

[index$_{32}$*1 - constExpr] { *<<additional address items inside "[]">>* }
[index$_{32}$*2 - constExpr] { *<<additional address items inside "[]">>* }
[index$_{32}$*4 - constExpr] { *<<additional address items inside "[]">>* }
[index$_{32}$*8 - constExpr] { *<<additional address items inside "[]">>* }

[base$_{32}$+index$_{32}$ - constExpr] { *<<additional address items inside "[]">>* }
[base$_{32}$+index$_{32}$*1 - constExpr] { *<<additional address items inside "[]">>*
}
[base$_{32}$+index$_{32}$*2 - constExpr] { *<<additional address items inside "[]">>*
}
[base$_{32}$+index$_{32}$*4 - constExpr] { *<<additional address items inside "[]">>*
}
[base$_{32}$+index$_{32}$*8 - constExpr] { *<<additional address items inside "[]">>*
}

The major restrictions is that there can be at most one base register (EAX, EBX, ECX, EDX, ESI, EDI, EBP, or ESP) and at most one index register (EAX, EBX, ECX, EDX, ESI, EDI, or EBP) in the address item. An optional static object name (**static, readonly**, or **storage** variable) or automatic variable name (**var** objects) may precede the address item list; however, keep in mind that if an automatic variable name precedes one of these bracketed expression lists, then the EBP register (or a user-defined register if the *reg$_{32}$::identifier* syntax is used) is already used as the base register. Here are some examples of legal addressing modes in HLA:

```
staticVar[ebx][ecx*4][ 4]
localVar[edi*2]
localVar[8][edx*8]
[ebx][edx+2]
```

Note that if you specify two 32-bit registers in an address expression without specifying an explicit scaled index value (e.g., "[ebx+ecx]") then HLA gets to choose which register is the base register and which is the index register (either choice will produce the correct effective address).

Any number of constant expressions inside brackets may appear in an extended address expression. HLA computes the sum of all such constant expressions and uses that sum as the single constant value. E.g.,

```
localVar[8][edx*8][2]  -- equivalent to -- localVar[edx*8 + 10]
```

16.3 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 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, dword**, **qword,** or **lword** to a register[1]; (2) when you need to assign an anonymous memory location a type. Here are a couple of examples:

```
if( (type int32 eax)  < 0 then

   inc( (type dword [ebx] ));

endif;
```

Type coercion is very useful in HLA when manipulating pointer objects, especially pointers to classes and records. Consider the following example:

```
type
   myRec_t: record
         i:int32;
         c:char;
   endrecord;

   mrPtr_t: pointer to myRec_t;

static
   mpr: mrPtr_t;
       .
       .
       .
   malloc( @size( myRec_t ) );
   mov( eax, mpr );
       .
       .
       .
   mov( mpr, ebx );
   mov( cl, (type myRec_t [ebx]).c );
   mov( 0, (type myRec_t [ebx]).i );
```

As you can see here, whatever memory address appears inside the parentheses is treated like an object of the specified type. So you can treat that whole entity as though it were a variable of the specified type (myRec_t in this example) and you can apply the dot operator or any other operation that would be legal on a variable of that type.

By default, the x86 general-purpose registers have the types **byte, word,** or **dword** (depending, of course, on their size). Sometimes you might want to coerce these registers to a different type, especially when outputting the value of a register or comparing a register with a constant. Coercion of a register is legal as long as the coerced data type is the same size as the register, e.g.,

```
(type int32 eax)
```

Coercion like this last example is especially useful when using the register without an output statement (like *stdout.put*) or in a run-time boolean expression. Consider the following:

```
if( eax < 0 ) then
   << do something if EAX is negative>>
endif;
```

1. 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.

In this example, the expression is always false because EAX is a **dword** object (which is unsigned). Therefore, EAX can never be less than zero (even if EAX contains something that you want interpreted as a negative value). You can solve this problem by coerce EAX to an **int32** object:

```
if( (type int32 eax) < 0 ) then
    << do something if EAX is negative>>
endif;
```

This code example will work properly since HLA is smart enough to generate the appropriate signed comparison/conditional jump sequence when it realizes one or more of the operands are signed.

Type coercion fully supports HLA memory addressing modes. You can use any valid HLA addressing mode form in place of the address object in the type coercion expression, for example:

```
    (type dword byteVar[ebx][ecx*1][2] )
```

In addition, you can also treat a type coercion operation as though it were a static identifier in an extended HLA addressing mode; that is, you can follow a type coercion operator with a set of bracketed addressing mode options:

```
    (type qword [ebx] )[ecx*8][16]
```