17 HLA v2.x Language Reference Manual

17.1 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.

A second difference, related to the fact that HLA uses a functional notation, is that HLA allows you to *compose* instructions. That is, one instruction may appear as an operand to a second instruction, e.g.,

```
mov( mov( 0, eax ), ebx );
```

To decipher this instruction, all you need to do is to realize that at compile time each instruction returns a string that HLA substitutes in place of the composed instruction. Usually, the string an instruction returns is that instruction’s destination operand. In the example above, the interior mov instruction’s destination operand is EAX, so that mov instruction "returns" the string "EAX" which HLA substitutes for the interior mov instruction, producing "mov(eax, ebx);" as the outside instruction. HLA always processes interior instructions from left-to-right interior-first. Therefore, the above instruction is equivalent to the MASM sequence:

```
    mov    eax, 0
    mov    ebx, eax
```

Consider a second example:

```
add( mov( i, eax ), mov( j, ebx ));
```

This instruction is equivalent to:

```
    mov    eax, i
    mov    ebx, j
    add    ebx, eax
```

Although, used sparingly, instruction composition is useful and can help improve the readability of your HLA programs in certain contexts, you should be careful when using instruction composition because it can quickly produce unreadable code. Even this second example (add(mov,mov)) would probably prove difficult to read by most programmers.

If you need to modify the RETURNS value of an instruction (in a macro, for example), you may use the "returns" statement in HLA. This statement takes the following form:

```
returns( { statements }, "string Constant" )
```

This statement emits the code for the statement(s) between the curly braces and then returns the specified string constant as the "returns" value for this statement.

The following paragraphs describe each of the HLA machine instructions. They also describe the string each instruction yields during compile time (this is called the "returns" string). Note that some instructions return the empty string as there is no return value one could reasonably associated with them. Such instructions cannot generally be used as operands within other instructions.

These descriptions do not describe the purpose for each instruction; see an assembly text like "The Art of Assembly Language Programming" for details on the operation of each instruction.

17.2 Zero Operand Instructions (Null Operand Instructions)

Instruction	Description
aaa( )	ASCII adjust for addition. Returns "ax".
aad( )	ASCII adjust for division. Returns "ax".
aam( )	ASCII adjust for multiplication. Returns "ax".
aas( )	ASCII adjust for subtraction. Returns "ax".
cbw( )	Convert byte to word (sign extension). Returns "ax"
cdq( )	Convert double to quadword. Returns "eax". Note: in the future, this may return "edx:eax".
clc( )	Clear carry flag. Returns "".
cld( )	Clear direction flag. Returns "".
cli( )	Clear interrupt flag. Returns "".
clts()	Clear task switched flag in CR0 (OS use only).
cmc( )	Complement carry flag. Returns "".
cmpsb( )	Compares the byte at [esi] to the byte at [edi] and increments or decrements ESI & EDI by one. Returns "".
cmpsd( )	Compares the dword at [esi] to the byte at [edi] and increments or decrements ESI & EDI by four. Returns "".
cmpsw( )	Compares the word at [esi] to the byte at [edi] and increments or decrements ESI & EDI by two. Returns "".
cpuid()	On entry, EAX contains zero, one, or two to determine how this instruction behaves. If EAX contains zero then this instruction returns vendor information in EAX, EBX, ECX, and EDX. If EAX contains one upon entry, EAX returns with version information and EDX contains feature information. If EAX contains two upon entry, EAX..EDX return with cache information. See the Intel documentation for more details concerning this instruction.
cwd( )	Convert word to doubleword. Returns "ax". Note: in the future, this may return "dx:ax".
cwde( )	Convert word to dword, extended. Returns "eax".
daa( )	Decimal adjust for addition. Returns "al".
das( )	Decimal adjust for subtraction. Returns "al".
hlt()	Halt instruction (OS and embedded use only).
insb( )	Inputs a byte from the port specified by DX and stores the byte at [EDI], then increments or decrements EDI by one. Returns "".

insd( )	Inputs a dword from the port specified by DX and stores the dword at [EDI], then increments or decrements EDI by four. Returns "".
insw( )	Inputs a word from the port specified by DX and stores the word at [EDI], then increments or decrements EDI by two. Returns "".
into( )	Interrupt on overflow. Returns "". Raises the ex.IntoInstr exception if the overflow flag is set when you execute this instruction.
invd()	Invalidate internal caches (OS use only).
iret( )	Interrupt return. Returns "".
iretd( )	Interrupt return poping 32-bit flags. Returns "".
lahf( )	Load AH from flags. Returns "al".
leave( )	Remove activation record from stack. Returns "".
lodsb( )	Load al from [ESI] and increment ESI by one. Returns "al".
lodsd( )	Load eax from [ESI] and increment ESI by four. Returns "eax".
lodsw( )	Load ax from [ESI] and increment ESI by two. Returns "ax".
movsb( )	Moves a byte from the location specified by [ESI] to the location specified by [EDI], then increments or decrements ESI & EDI by one. Returns "".
movsd( )	Moves a dword from the location specified by [ESI] to the location specified by [EDI], then increments or decrements ESI & EDI by four. Returns "".
movsw( )	Moves a word from the location specified by [ESI] to the location specified by [EDI], then increments or decrements ESI & EDI by two. Returns "".
nop( )	No operation. Returns "".
outsb( )	Outputs the byte at address [ESI] to the port specified by DX, then increments or decrements ESI by one. Returns "".
outsd( )	Outputs the dword at address [ESI] to the port specified by DX, then increments or decrements ESI by four. Returns "".
outsw( )	Outputs the word at address [ESI] to the port specified by DX, then increments or decrements ESI by two. Returns "".
popad( )	Pop all general purpose 32-bit registers from stack. Returns "".
popa( )	Pop all general purpose 16-bit registers from stack. Returns "".
popf( )	Pop 16-bit flags register from stack. Returns "".
popfd( )	Pop 32-bit flags register from stack. Returns "".
pusha( )	Push all general-purpose 16-bit registers onto the stack. Returns "".
pushad( )	Push all general-purpose 32-bit registers onto the stack. Returns "".
pushf( )	Push 16-bit flags register onto the stack. Returns "".
pushfd( )	Push 32-bit flags register onto the stack. Returns "".

rdmsr()	Read from model specific register specified by ECX into EDX:EAX (OS use only).
rdpmc()	Read performance monitoring counter specified by ECX into EDX:EAX (OS use only).
rdtsc()	Reads the "time stamp" counter and returns the 64-bit result in edx:eax.
rep.insb( )	Transfers ECX bytes from the port specified by DX to the location specified by [EDI]. Increments or decrements EDI by one after each transfer. Returns "".
rep.insd( )	Transfers ECX dwords from the port specified by DX to the location specified by [EDI]. Increments or decrements EDI by four after each transfer. Returns "".
rep.insw( )	Transfers ECX words from the port specified by DX to the location specified by [EDI]. Increments or decrements EDI by two after each transfer. Returns "".
rep.movsb( )	Copies ECX bytes from the memory location specified by [ESI] to the location specified by [EDI]. Increments or decrements EDI & ESI by one after each transfer. Returns "".
rep.movsd( )	Copies ECX dwords from the memory location specified by [ESI] to the location specified by [EDI]. Increments or decrements EDI & ESI by four after each transfer. Returns "".
rep.movsw( )	Copies ECX words from the memory location specified by [ESI] to the location specified by [EDI]. Increments or decrements EDI & ESI by two after each transfer. Returns "".
rep.outsb( )	Transfers ECX bytes from the location specified by [ESI] to the port specified by DX. Increments or decrements EDI by one after each transfer. Returns "".
rep.outsd( )	Transfers ECX dwords from the location specified by [ESI] to the port specified by DX. Increments or decrements EDI by four after each transfer. Returns "".
rep.outsw( )	Transfers ECX words from the location specified by [ESI] to the port specified by DX. Increments or decrements EDI by two after each transfer. Returns "".
rep.stosb( )	Copies CX bytes from AL to the location specified by [EDI]. Increments or decrements EDI by one after each transfer. Returns "".
rep.stosd( )	Copies ECX dwords from EAX to the location specified by [EDI]. Increments or decrements EDI by four after each transfer. Returns "".
rep.stosw( )	Copies ECX words from AX to the location specified by [EDI]. Increments or decrements EDI by two after each transfer. Returns "".
repe.cmpsb( )	Compares ECX bytes starting at location [ESI] to the set of bytes at location [EDI] as long as the bytes are equal. The comparison stops once two unequal bytes are found. After each successful compare, this instruction increments or decrements ESI and EDI by one (and decrements ECX). Returns "".
repe.cmpsd( )	Compares ECX dwords starting at location [ESI] to the set of dwords at location [EDI] as long as the dwords are equal. The comparison stops once two unequal dwords are found. After each successful compare, this instruction increments or decrements ESI and EDI by four (and decrements ECX). Returns "".
repe.cmpsw( )	Compares ECX words starting at location [ESI] to the set of words at location [EDI] as long as the words are equal. The comparison stops once two unequal words are found. After each successful compare, this instruction increments or decrements ESI and EDI by two (and decrements ECX). Returns "".

repe.scasb( )	Compares AL against ECX bytes starting at location [EDI] as long as the bytes are equal. The comparison stops once two unequal bytes are found. After each successful compare, this instruction increments or decrements EDI by one (and decrements ECX). Returns "".
repe.scasd( )	Compares EAX against ECX dwords starting at location [EDI] as long as the dwords are equal. The comparison stops once two unequal dwords are found. After each successful compare, this instruction increments or decrements EDI by four (and decrements ECX). Returns "".
repe.scasw( )	Compares AX against ECX words starting at location [EDI] as long as the words are equal. The comparison stops once two unequal words are found. After each successful compare, this instruction increments or decrements EDI by two (and decrements ECX). Returns "".
repne.cmpsb( )	Compares ECX bytes starting at location [ESI] to the set of bytes at location [EDI] as long as the bytes are not equal. The comparison stops once two equal bytes are found. After each successful compare, this instruction increments or decrements ESI and EDI by one (and decrements ECX). Returns "".
repne.cmpsd( )	Compares ECX dwords starting at location [ESI] to the set of dwords at location [EDI] as long as the dwords are not equal. The comparison stops once two equal dwords are found. After each successful compare, this instruction increments or decrements ESI and EDI by four (and decrements ECX). Returns "".
repne.cmpsw( )	Compares ECX words starting at location [ESI] to the set of words at location [EDI] as long as the words are not equal. The comparison stops once two equal words are found. After each successful compare, this instruction increments or decrements ESI and EDI by two (and decrements ECX). Returns "".
repne.scasb( )	Compares AL against ECX bytes starting at location [EDI] as long as the bytes are not equal. The comparison stops once two equal bytes are found. After each successful compare, this instruction increments or decrements EDI by one (and decrements ECX). Returns "".
repne.scasd( )	Compares EAX against ECX dwords starting at location [EDI] as long as the dwords are not equal. The comparison stops once two equal dwords are found. After each successful compare, this instruction increments or decrements EDI by four (and decrements ECX). Returns "".
repne.scasw( )	Compares AX against ECX words starting at location [EDI] as long as the words are not equal. The comparison stops once two equal words are found. After each successful compare, this instruction increments or decrements EDI by two (and decrements ECX). Returns "".
rsm()	Resume from system management mode (OS use only).
sahf( )	Store AH into the flags register. Returns "ah".
scasb( )	Compares the byte in al to the location specified by [EDI], then increments or decrements EDI by one. Returns "".
scasd( )	Compares the dword in eax to the location specified by [EDI], then increments or decrements EDI by four. Returns "".
scasw( )	Compares the word in ax to the location specified by [EDI], then increments or decrements EDI by two. Returns "".

stc( )	Set the carry flag. Returns "".
std( )	Set the direction flag. Returns "".
sti( )	Set the interrupt flag. Returns "".
stosb( )	Stores the byte in al to the location specified by [EDI], then increments or decrements EDI by one. Returns "".
stosd( )	Stores the dword in eax to the location specified by [EDI], then increments or decrements EDI by four. Returns "".
stosw( )	Stores the word in ax to the location specified by [EDI], then increments or decrements EDI by two. Returns "".
ud2()	Undefined opcode instruction. This instruction always raises an undefine opcode exception.
wbinvd()	Write back and invalidate cache (OS use only).
wait( )	Coprocessor wait instruction. Returns "".
xlat( )	Translate instruction. Returns "".

Note: if the NULL-Operand instructions appear as a stand-alone instruction (i.e., they are not part of an instruction composition and, thus, appear as the operand to another instruction), you can drop the "( )" after the instruction as long as you terminate the instruction with a semicolon.

17.3 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);
lock.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]));"

These instructions all return their destination operand as the "returns" value.

See "The Art of Assembly" for a further discussion of these instructions.

If the "lock." prefix is present, the instruction asserts the bus lock signal during execution. The "lock." prefix is valid only on instructions that reference memory.

17.4 The XCHG Instruction

The xchg instruction allows the following syntactical forms:

Generic Form:

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

This instruction returns its destination operand as its "returns" value.

If the "lock." prefix is present, the instruction asserts the bus lock signal during execution. The "lock." prefix is valid only on instructions that reference memory.

17.5 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);".

These instructions return their dest (first) operand as their "returns" value.

17.6 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.

These instructions return their destination operand (AX, DX:AX, or EDX:EAX for the extended precision MUL and IMUL instructions) as their "returns" value.

See "The Art of Assembly Language Programming" for more details on these instructions.

17.7 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.

The DIV and IDIV instructions return "AL", "AX", or "EAX" as their "returns" value (the quotient is left in the accumulator register). The MOD and IMOD instructions return "AH", "DX", or "EDX" as their "returns" value. Indeed, the "returns" value is the only difference between these instructions. The DIV and MOD instructions compile into the 80x86 DIV instruction; the IDIV and IMOD instructions compile into the 80x86 IDIV instruction.

See the "Art of Assembly" for a further discussion of these instructions.

17.8 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 );;
    lock.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]));"

These instructions all return their destination operand as the "returns" value.

See the "Art of Assembly" for a further discussion of these instructions.

If the "lock." prefix is present, the instruction asserts the bus lock signal during execution. The "lock." prefix is valid only on instructions that reference memory.

17.9 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]));"

These instructions return their destination operand as their "returns" value.

See the "Art of Assembly" for a further discussion of these instructions.

17.10The 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 )
```

These instructions return their destination operand as the "returns" value.

See the "Art of Assembly" for a further discussion of these instructions.

17.11 The Lea Instruction

These instructions use the following syntax:

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

Extended Syntax:

```
lea( Reg32, StringConstant )
lea( Reg32, const ConstExpr )

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 that 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.

The extended syntax form lets you specify a constant rather than a memory address. There is no such thing as the address of a constant, but HLA will create a memory variable in the constants data segment and initialize that variable with the value of the specified memory constant and then load the address of this variable into the specified register (or push it onto the stack).

There is a subtle difference between the following two instructions:

```
lea( eax, "String" );
lea( eax, const "String" );
```

The first instruction loads EAX with the address of the first character of the literal string constant. The second form loads the EAX register with the address of a string variable (which is a pointer containing the address of the first character of the string literal).

The LEA instructions return the 32-bit register as their "returns" value.

See "The Art of Assembly" for a further discussion of the LEA instruction.

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.

17.12The 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. They return their destination operand as their "returns" value.

See the "Art of Assembly" for a further discussion of these instructions.

17.13The 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. They all return their operand as their "returns" value.

17.14Procedure Calls

HLA provides several different ways to call a procedure. Given a procedure named "MyProc", any of the following syntaxes are legal:

```
MyProc( parameter_list );
call( MyProc );
call MyProc;
```

If MyProc has a set of declared parameters, the number and types of actual parameters must match the number and types of the formal parameters. HLA will emit the code needed to push the parameter list on the stack. In the two call statements above, it is the programmer's responsibility to pass any needed parameters. For more details, see the section on procedure declarations.

In the examples above, `MyProc` can either be the name of an actual procedure or a procedure variable (that is a pointer to a procedure declared as "myproc:procedure(*parameters*);" in the VAR or a static section). If you need to call a procedure using an anonymous memory variable (i.e., an addressing mode like [ebx]), an untyped dword value, or via a register, you must use the syntax of the second call above, e.g., "call(ebx);". Of course, any legal HLA/80x86 address mode would be legal here.

When declaring a standard procedure, the procedure declaration syntax allows you to specify a "returns" value for that procedure, e.g.,

```
procedure MyProc; returns( "eax" );
```

HLA substitutes the string that appears as the "returns" argument for the call when using the first syntax above. For example, supposing that `MyProc` is a function returning its result in EAX, you could use the following to call `MyProc` and save the return value in the "Result" variable:

```
mov( MyProc(), Result );
```

For more details, see the section on procedure declarations.

To call a class procedure, one would use one of the following syntaxes:

```
className.ProcName( parameters );
call( className.ProcName );
call ClassName.ProcName;


objectName.ProcName( parameters );
call( objectName.ProcName );
call objectName.ProcName;
```

The difference between "`className`" and "`objectName`" is that "`className`" represents the actual name of the class data type whereas "`objectName`" represents the name of an instance of this class (i.e., a variable of type "`className`" declared in the VAR or a static section).

When calling a class procedure, HLA loads the ESI register with the address of the object before calling the specified procedure. Since there is no instance variable (object) associated with the `className` form, HLA loads ESI with zero (NULL). Inside the class procedure, you can test the value of ESI to determine if the procedure was called via the class name or an object name. This is quite useful, for example when writing constructors, to determine whether the procedure needs to allocate storage for an object. Consider the following program that demonstrates the use of an object constructor (create):

```
program demo;

#include( "memory.hhf" );
#include( "stdio.hhf" );
```

```
type
   cc: class

         var
            i:int32;

         procedure create;  returns( "esi" );

      endclass;


var
   ccVar: cc;
   ccPtr: pointer to cc;

static
   ccStat:cc;

procedure cc.create; @nodisplay;
begin create;

   push( eax );
   if( esi = 0 ) then

      stdout.put( "Allocating" nl );
      malloc( @size( cc ));
      mov( eax, esi );

   else

      stdout.put( "Already allocated" nl );

   endif;
   mov( &cc._VMT_, this._pVMT_ );
   mov( 0, this.i );
   pop( eax );

end create;

begin demo;

   // This first call to create allocates storage.

   mov( cc.create(), ccPtr );

   // In all the remaining calls, ESI is loaded with
   // the address of the object and no storage is
   // created.

   ccPtr.create();
   ccVar.create();
   ccStat.create();

end demo;
```

The call( ) statement allows any one of the following syntaxes:

```
call ProcID;
call( ProcID );
call( dwordvar );
call( anonmem );        // Addressing mode like [ebx].
call( Reg32 );
```

The second form above returns the string (if any) specified by ProcID's "returns" option. The remaining call instructions return the empty string as their "returns" value.

You may also call an iterator procedure via the CALL instruction. However, it is your responsibility to set up the parameters and other state information prior to the call (see the section on iterators for more details).

17.15The 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).

Normally, you would use these instructions in a procedure that has the "@noframe" option. Unless you know exactly what you are doing, you should never use the "RET" instruction inside a standard HLA procedure without this option since doing so almost always produces disastrous results. If you do use this instruction within such a procedure, it is your responsibility to deallocate local variables and the display (if any), restore EBP, and remove any parameters from the stack.

17.16The 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.

17.17The 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 (or a globally visible label declared in a label section or a global label defined with the "::" symbol).

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

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 if you are emitting assembly language source (rather than directly emitting object code). Unlike the other conditional jump instructions, these instructions have a very limited +/- 128-byte range. Unfortunately, HLA cannot detect if the branch is out of range (this task is handled by back-end assembler when producing assembly language source code), so if a range error occurs, HLA cannot warn you about this. The 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.

In a few special cases, the boolean constants "true" and "false" are legal labels. See the discussion of HLA's high-level language features for more details.

17.18The 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 )
```

See the "Art of Assembly" for a further discussion of these instructions.

17.19The 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.

17.20The 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. The IN instructions return the accumulator register (AL, AX, or EAX) as their "returns" value. The OUT instructions return the port number (or DX) as their "returns" value.

Note that these instructions may be privileged instructions when running under Win32 or *NIX. Their use may generate a fault in certain instances or when accessing certain ports.

See the "Art of Assembly" for a further discussion of these instructions.

17.21The 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 *NIX calls.

17.22Bound 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 )
```

These instructions return the register as their "returns" value.

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.

The BOUND instruction compares the register operand against the two constants (or the two consecutive memory locations at the specified address). If the register value is outside the range specified by the operand(s), then the 80x86 CPU raises an ex.BoundInstr exception. You can handle this exception using the TRY..ENDTRY HLL statement in HLA.

Because the BOUND instruction tends to be slow, and of course it consumes memory, many programmers don't use it as often as they should for fear it will make their programs less efficient. HLA solves this problem using the "@bound" compile-time pseudo-variable. If @bound contains true (the default value) then HLA will compile the BOUND instruction and it will behave normally. If @bound contains false, then HLA will not emit any code for the bound instruction (this is similar to "asserts" in C/C++). You can set the value of @bound in the VAL section or with the "?" operator, e.g.,

```
?@bound := false;

// Code that ignores BOUND instructions
    .
    .
    .
?@bound := true;

// BOUND instructions are active again.
```

17.23The 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 "Art of Assembly" for a further discussion of this instruction and the LEAVE instruction.

17.24CMPXCHG Instruction

This instruction uses the following syntax:

Generic Form:

```
cmpxchg( reg/mem, reg );
lock.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 )
```

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

See the "Art of Assembly" for a further discussion of this instruction.

If the "lock." prefix is present, the instruction asserts the bus lock signal during execution. The "lock." prefix is valid only on instructions that reference memory.

17.25CMPXCHG8B Instruction

This instruction uses the following syntax:

Generic Form:

```
cmpxchg( mem64 );
lock.cmpxchg8b( mem64);
```

This instruction compares edx:eax with the specified qword operand. If the values are equal, this instruction stores the value in ECX:EBX into the destination operand; otherwise it loads the memory operand into EDX:EAX.

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

See the "Art of Assembly" for a further discussion of this instruction.

If the "lock." prefix is present, the instruction asserts the bus lock signal during execution. The "lock." prefix is valid only on instructions that reference memory.

17.26The XADD Instruction

The XADD instruction uses the following syntax:

Generic Form:

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

This instruction returns its destination operand as its "returns" value.

See the "Art of Assembly" for a further discussion of this instruction.

If the "lock." prefix is present, the instruction asserts the bus lock signal during execution. The "lock." prefix is valid only on instructions that reference memory.

17.27BSF 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 );
```

These instructions return the destination register as their "returns" value.

See the "Art of Assembly" for a further discussion of these instructions.

17.28The 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.

It returns the 32-bit register as its "returns" value.

See the "Art of Assembly" for a further discussion of this instruction.

17.29Bit 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 );

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

Substitute the BTC, BTR, or BTS mnemonic for BT in the examples above for these other instructions. The BTC, BTR, and BTS instructions also allow a "lock." prefix, e.g., "lock.btc(reg32, mem);" If the "lock." prefix is present, the instruction asserts the bus lock signal during execution. The "lock." prefix is valid only on instructions that reference memory.

These instructions return the destination operand as their "returns" value.

Notice the two special forms that allow character set variables. HLA actually casts these 16-byte objects as word or dword memory variables, but they otherwise work just fine with cset objects.

Special forms available only with the BT instruction:

```
bt( reg16, CharacterSetConstant );
bt( reg32, CharacterSetConstant );
```

These two forms return the source register (BitNumber) as their "returns" value. Note that HLA will create a phantom variable that contains the character set constant and then supplies the name of this constant, effectively making these two instruction equivalent to "bt(reg, CharacterSetVariable);".

See the "Art of Assembly" for a further discussion of these instructions.

17.30Floating Point Instructions

HLA supports the following FPU instructions. Note: all FPU instructions have a "returns" value of "st0" unless otherwise noted.

```
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. HLA v2.0 actually supports the entire FPU instruction set. See the Intel documentation for more details.

17.31 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.

17.32 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

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();
```

The following MMX shift instructions also require a special syntax. They allow the following two forms:

```
mmxshift( immConst, mmxReg );
mmxshift( mmxReg, mmxReg );

 psllw
 pslld
 psllq
 psrlw
 psrld
 psrlq
 psraw
 psrad
```

Note that the `psllw, psrlw,` and `psraw` instructions only allow an immediate constant in the range 0..15, the `pslld, psrld,` and `psrad` instructions only allow constants in the range 0..31, the `psllq` and `psrlq` instructions only allow immediate constants in the range 0..63.

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

17.33SSE Instructions

HLA supports the following SSE and SSE/2 instructions found on the Pentium III, IV, and later processors (note that some instructions are only available on Pentium IV and later processors; see the Intel reference manuals for details):

HLA uses the symbols xmm0, xmm1, ..., xmm7 for the SSE register set.

SSE Instrs:

```
addsd( sseReg/mem128, sseReg );
addpd( sseReg/mem128, sseReg );
addps( sseReg/mem128, sseReg );
addss( sseReg/mem128, sseReg );
andnpd( sseReg/mem128, sseReg );
andnps( sseReg/mem128, sseReg );
andpd( sseReg/mem128, sseReg );
andps( sseReg/mem128, sseReg );

clflush( mem8 );

cmppd( imm8, sseReg/mem128, sseReg );
cmpps( imm8, sseReg/mem128, sseReg );
cmpsdp( imm8, sseReg/mem64, sseReg );
cmpss( imm8, sseReg/mem32, sseReg );
cmpeqss( sseReg, sseReg );
cmpltss( sseReg, sseReg );
cmpless( sseReg, sseReg );
cmpneqss( sseReg, sseReg );
cmpnlts( sseReg, sseReg );
cmpnles( sseReg, sseReg );
```

```
cmpords( sseReg, sseReg );
cmpunordss( sseReg, sseReg );
cmpeqsd( sseReg, sseReg );
cmpltsd( sseReg, sseReg );
cmplesd( sseReg, sseReg );
cmpneqsd( sseReg, sseReg );
cmpnlts( sseReg, sseReg );
cmpnles( sseReg, sseReg );
cmpords( sseReg, sseReg );
cmpunords( sseReg, sseReg );

cmpeqps( sseReg, sseReg );
cmpltps( sseReg, sseReg );
cmpleps( sseReg, sseReg );
cmpneqps( sseReg, sseReg );
cmpnltp( sseReg, sseReg );
cmpnleps( sseReg, sseReg );
cmpordps( sseReg, sseReg );
cmpunordps( sseReg, sseReg );

cmpeqpd( sseReg, sseReg );
cmpltpd( sseReg, sseReg );
cmplepd( sseReg, sseReg );
cmpneqpd( sseReg, sseReg );
cmpnltpd( sseReg, sseReg );
cmpnlepd( sseReg, sseReg );
cmpordpd( sseReg, sseReg );
cmpunordpd( sseReg, sseReg );

comisd( sseReg/mem64, sseReg );
comiss( sseReg/mem32, sseReg );
cvtdq2pd( sseReg/mem64, sseReg );
cvtdq2pq
cvtdq2ps( sseReg/mem128, sseReg );
cvtpd2dq( sseReg/mem128, sseReg );
cvtpd2pi( sseReg/mem128, mmxReg );
cvtpd2ps( sseReg/mem128, sseReg );
cvtpi2pd( sseReg/mem64, sseReg );
cvtpi2ps( sseReg/mem64, sseReg );
cvtpi2ss
cvtps2dq( sseReg/mem128, sseReg );
cvtps2pd( sseReg/mem64, sseReg );
cvtps2pi( sseReg/mem64, sseReg );
cvtsd2si( sseReg/mem64, Reg32 );
cvtsi2sd( Reg32/mem32, sseReg );
cvtsi2ss( sseReg/mem64, sseReg );
cvtss2sd( sseReg/mem32, sseReg );
cvtsd2ss( Reg32/mem32, sseReg );
cvtss2si( sseReg/mem32, Reg32 );
cvttpd2pi( sseReg/mem128, mmxReg );
cvttpd2dq( sseReg/mem128, sseReg );
cvttps2dq( sseReg/mem128, sseReg );
cvttps2pi( sseReg/mem64, mmxReg );
cvttsd2si( sseReg/mem64, Reg32 );
cvttss2si( sseReg/mem32, Reg32 );

divpd( sseReg/mem128, sseReg );
```

```
divps( sseReg/mem128, sseReg );
divsd( sseReg/mem64, sseReg );
divss( sseReg/mem32, sseReg );
fxsave( mem512 );
fxrstor( mem512 );
ldmxcsr( mem32 );
lfence

maskmovdqu( sseReg, sseReg );
maskmovq( mmxReg, mmxReg );
maxpd( sseReg/mem128, sseReg );
maxps( sseReg/mem128, sseReg );
maxsd( sseReg/mem64, sseReg );
maxss( sseReg/mem32, sseReg );

mfence

minpd( sseReg/mem128, sseReg );
minps( sseReg/mem128, sseReg );
minsd( sseReg/mem64, sseReg );
minss( sseReg/mem32, sseReg );

movapd( sseReg/mem128, sseReg );
movapd( sseReg, sseReg/mem128 );
movaps( sseReg/mem128, sseReg );
movaps( sseReg, sseReg/mem128 );
movdqa( sseReg/mem128, sseReg );
movdqa( sseReg, sseReg/mem128 );
movdqu( sseReg/mem128, sseReg );
movdqu( sseReg, sseReg/mem128 );
movdq2q( sseReg, mmxReg );
movhlps( sseReg, sseReg );
movhpd( mem64, sseReg );
movhpd( sseReg, mem64 );
movhps( mem64, sseReg );
movhps( sseReg, mem64 );
movlpd( mem64, sseReg );
movlpd( sseReg, mem64 );
movlps( mem64, sseReg );
movlps( sseReg, mem64 );
movlhps( sseReg, sseReg );
movmskpd( sseReg, Reg32 );
movmskps( sseReg, Reg32 );
movnti( Reg32, mem32 );
movntpd( sseReg, mem128 );
movntps( sseReg, mem128 );
movntq( mmxReg, mem64 );
movntdq( sseReg, mem128 );
movq2dq( mmxReg, sseReg );
movsdp( sseReg, sseReg );
movsdp( mem64, sseReg );
movsdp( sseReg, mem64 );
movss( sseReg, sseReg );
movss( mem32, sseReg );
movss( sseReg, mem32 );
movupd( sseReg, sseReg );
movupd( sseReg, mem128 );
movupd( mem128, sseReg );
movups( sseReg, sseReg );
movups( sseReg, mem128 );
movups( mem128, sseReg );

mulpd( sseReg/mem128, sseReg );
mulps( sseReg/mem128, sseReg );
mulss( sseReg/mem32, sseReg );
mulsd( sseReg/mem64, sseReg );

orpd( sseReg/mem128, sseReg );
orps( sseReg/mem128, sseReg );

pause

pmuludq( mmxReg/mem64, mmxReg );
pmuludq( sseReg/mem128, sseReg );

prefetcht0( mem8 );
prefetcht1( mem8 );
prefetcht2( mem8 );
prefetchnta( mem8 );

pshufd( imm8, sseReg/mem128, sseReg );
pslldq( imm8, sseReg );
psrldq( imm8, sseReg );
punpckhqdq( sseReg/mem128, sseReg );
punpcklqdq( sseReg/mem128, sseReg );

rcpps( sseReg/mem128, sseReg );
rcpss( sseReg/mem128, sseReg );
rsqrtps( sseReg/mem128, sseReg );
rsqrtss( sseReg/mem32, sseReg );

sfence;

shufpd( imm8, sseReg/mem128, sseReg );
shufps( imm8, sseReg/mem128, sseReg );
sqrtpd( sseReg/mem128, sseReg );
sqrtps( sseReg/mem128, sseReg );
sqrtsd( sseReg/mem64, sseReg );
sqrtss( sseReg/mem32, sseReg );

stmxcsr( mem32 );

subps( sseReg/mem128, sseReg );
subpd( sseReg/mem128, sseReg );
subsd( sseReg/mem64, sseReg );
subss( sseReg/mem32, sseReg );

ucomisd( sseReg/mem64, sseReg );
ucomiss( sseReg/mem32, sseReg );

unpckhpd( sseReg/mem128, sseReg );
unpckhps( sseReg/mem128, sseReg );
unpcklpd( sseReg/mem128, sseReg );
unpcklps( sseReg/mem128, sseReg );
```

```
xorpd( sseReg/mem128, sseReg );
xorps( sseReg/mem128, sseReg );
```

17.34OS/Priviledged Mode Instructions

Although HLA was originally intended for writing 32-bit flat model user mode applications, some HLA users may wish to write an operaing system kernel or device drivers within HLA. Therefore, HLA provides support for various priviledged instructions and instructions that manipulate segment registers on the 80x86 processor. This section describes those instructions. Normal application programs should not use these instructions (most will cause a "General Protection Fault" if you attempt to execute them).

For additional information on these instructions, please see the Intel documentation for the Pentia processors.

```
arpl( r16, r/m16 );
```

Adjusts the RPL field of a segment descriptor.

```
clts();
```

Clears the task switched flag in CR0.

```
hlt();
```

Halts the processor until an interrupt or reset comes along.

```
invd();
```

Invalidates the internal cache.

```
invlpg( mem );
```

Invalidates the TLB entry associated with the memory address specified as the source operand.

```
lar( r/m16, r16 );
lar( r/m32, r32 );
```

Load access rights from the segment descriptor specified by the first operand into the second operand.

```
lds( r32, m48 );
les( r32, m48 );
lfs( r32, m48 );
lgs( r32, m48 );
lss( r32, m48 );
```

Load a far (48-bit) segmented pointer into ds, es, fs, gs, or ss, and some other 32-bit register. Note that HLA does not support an `fword` data type. These instructions require a 48-bit memory operand, nonetheless. You may create your own 48-bit `fword` data type using a record declaration like the following:

```
type
   fword: record
      offset: dword;
```

```
        selector: word;
    endrecord;


lgdt( mem48 );
lidt( mem48 );
sgdt( mem48 );
sidt( mem48 );
```

Loads or stores the global descriptor table pointer (lgdt/sgdt) or interrupt descriptor table pointer (lidt/sidt) via the specified 48-bit memory operand. HLA does not support a 48-bit data type specifically for these instructions, but you can easily create one as follows:

```
type
    descPtr: record
        lowerLimit: word;
        baseAdrs: dword;
    endrecord
```

```
lldt( r/m16 );
sldt( r/m16 )
```

These instructions copy the specified source operand to/from the local descriptor table.

```
lsl( r/m16, r16 );
lsl( r/m32, r32 );
```

Load segment limit instruction;

```
ltreg( r/m16 );
streg( r/m16 );
```

Load and store the task register. Note that Intel uses the mnemonics "ltr" and "str" for these instructions. HLA changes these mnemonics to avoid conflicts with the commonly used "str" namespace (the HLA strings module).

```
mov( r/m16, segreg );
mov( segreg, r/m16 );
```

Copies data between an 80x86 segment register and a 16-bit register or memory location. Note that HLA uses the following register names for the segment registers:

cseg	The 80x86 CS register.
dseg	The 80x86 DS register
eseg	The 80x86 ES register
fseg	The 80x86 FS register
gseg	The 80x86 GS register
sseg	The 80x86 SS register

HLA uses these names rather than the Intel standard register names to avoid conflicts with the "cs" (cset) namespace identifier and other commonly used application identifiers. Note that `CSEG` may not be a destination register for the `MOV` instruction.

```
mov( r32, crx );      // note: x= 0, 2, 3, or 4.
mov( crx, r32 );
```

These instructions move data between one of the 32-bit registers and one of the x86's control registers. Note that HLA reserves names cr0..cr7 even though Intel doesn't currently define all eight control registers.

```
mov( r32, drx );      // note: x=0, 1, 2, 3, 6, 7
mov( drx, r32 );
```

These instructions move data between the general-purpose 32-bit registers the the x86 debug registers. Note that HLA reserves names dr0..dr7 even though the assembler doesn't currently support the user of the dr4 and dr5 registers.

```
push( segreg );
pop( segreg );
```

These instructions push and pop the x86 segment registers (cseg, dseg, eseg, fseg, gseg, and sseg). Note, however, that you cannot pop the cseg register. (see the comment earlier about HLA segment register names).

```
rdmsr();
rdpmc();
```

These instructions read model-specific registers or performance monitoring registers on the x86. The ECX register specifies the register to read, these instructions copy the data to EDX:EAX.

```
rsm();
```

Resumes from system management mode.

```
verr( r/m16 );
verw( r/m16 );
```

Verifies whether the specified code segment is readable (verr) or writable (verw) from the current priviledge level.

```
wbinvd();
```

Write-back and invalidate cache.

17.35Other Instructions and features

Currently, HLA does not support 3DNow, or certain other SIMD instructions found on later x86 processors. The intent is to add support in the near future.

Note that HLA does not support the LMSW and SMSW instructions (old, obsolete 286 instructions). Use MOV with CR0 instead.

In the meantime, if you need to use any of these instructions you can use the #ASM..#ENDASM and #EMIT directives to insert them into your programs. You can also use macros to implement any desired instructions or syntaxes you desire.

Segment overrides are possible using the segment names (cseg, dseg, eseg, fseg, gseg, and sseg) as a label before an instruction, e.g.,

```
fseg: mov( [eax], eax );  // Fetches from fs:[eax].
```

Generally, you don't need segment overrides in flat-model 32-bit OS environments. However, the operating system kernel (even flat-model OSes) sometimes need to apply a segment override, for example to support Structured Exception Handling under Windows, hence this discussion.