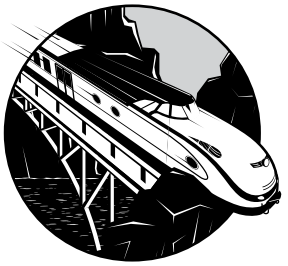
A

THE MINIMAL 80X86 INSTRUCTION SET



Although the 80x86 CPU family supports hundreds of instructions, few compilers actually use more than a couple dozen of them. This is because many instructions have become obsolete over time as newer instructions have emerged. Some instructions, such as many of the Pentium's MMX and SSE instructions, simply don't correspond to functions you'd normally perform in an HLL. As a result, compilers rarely generate these types of machine instructions, which generally appear only in hand-written assembly language programs. Fortunately, this means you don't need to learn the entire 80x86 instruction set in order to study compiler output, but only the handful that compilers typically emit. This appendix covers that subset.

A.1 add

The add instruction requires two operands: a source and a destination. It computes the sum of the values of these two operands and stores the sum back into the destination operand. It also sets several flags in the EFLAGS register, based on the result of the addition operation.

Table A-1: HLA Syntax for add

Instruction	Description
add(*constant*, *destreg*);	*destreg* := *destreg* + *constant* *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
add(*constant*, *destmem*);	*destmem* := *destmem* + *constant* *destmem* may be any 8-bit, 16-bit, or 32-bit memory variable.
add(*srcreg*, *destreg*);	*destreg* := *destreg* + *srcreg* *destreg* and *srcreg* may be any 8-bit, 16-bit, or 32-bit general-purpose registers, though they must both be the same size.
add(*srcmem*, *destreg*);	*destreg* := *destreg* + *srcmem* *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register; *srcmem* can be any like-sized memory location.
add(*srcreg*, *destmem*);	*destmem* := *destmem* + *srcreg* *srcreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register; *destmem* can be any like-sized memory location.

Table A-2: Gas Syntax for add

Instruction	Description

addb *constant*, $destreg_8$
addw *constant*, $destreg_{16}$
addl *constant*, $destreg_{32}$
	$destreg_n$:= $destreg_n$ + *constant*
	$destreg_n$ may be any 8-bit, 16-bit, or 32-bit general-purpose register, as appropriate for the suffix.

addb *constant*, $destmem_8$
addw *constant*, $destmem_{16}$
addl *constant*, $destmem_{32}$
	$destmem_n$:= $destmem_n$ + *constant*
	$destmem_n$ may be any 8-bit, 16-bit, or 32-bit memory variable, as appropriate for the suffix.

addb $srcreg_8$, $destreg_8$
addw $srcreg_{16}$, $destreg_{16}$
addl $srcreg_{32}$, $destreg_{32}$
addq $srcreg_{64}$, $destreg_{64}$
	$destreg_n$:= $destreg_n$ + $srcreg_n$
	$destreg_n$ and $srcreg_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose registers, as specified by the suffix.

addb $srcmem_8$, $destreg_8$
addw $srcmem_{16}$, $destreg_{16}$
addl $srcmem_{32}$, $destreg_{32}$
addq $srcmem_{64}$, $destreg_{64}$
	$destreg_n$:= $destreg_n$ + $srcmem_n$
	$destreg_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, according to the suffix; $srcmem_n$ can be any like-sized memory location.

addb $srcreg_8$, $destmem_8$
addw $srcreg_{16}$, $destmem_{16}$
addl $srcreg_{32}$, $destmem_{32}$
addq $srcreg_{64}$, $destmem_{64}$
	$destmem_n$:= $destmem_n$ + $srcreg_n$
	$srcreg_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, as specified by the suffix; $destmem_n$ can be any like-sized memory location.

Table A-3: MASM Syntax for add

Instruction	Description
add *destreg*, *constant*	*destreg* := *destreg* + *constant* *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
add *destmem*, *constant*	*destmem* := *destmem* + *constant* *destmem* may be any 8-bit, 16-bit, or 32-bit memory variable.
add *destreg*, *srcreg*	*destreg* := *destreg* + *srcreg* *destreg* and *srcreg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose registers, though they must both be the same size.
add *destreg*, *srcmem*	*destreg* := *destreg* + *srcmem* *destreg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register; *srcmem* can be any like-sized memory location.
add *destmem*, *srcreg*	*destmem* := *destmem* + *srcreg* *srcreg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register; *destmem* can be any like-sized memory location.

Table A-4: EFLAGS Settings for add

Flag	Setting
Carry	Set if the sum of the two values produces an unsigned overflow.
Overflow	Set if the sum of the two values produces a signed overflow.
Sign	Set if the sum of the two values has a 1 in its HO bit position.
Zero	Set if the sum of the two values is 0.

A.2 and

The and instruction requires two operands: a source and a destination. It computes the bitwise logical AND of the values of these two operands and stores the result back into the destination operand. It also sets several flags in the EFLAGS register, based on the result of the bitwise AND operation.

Table A-5: HLA Syntax for and

Instruction	Description
and(*constant*, *destreg*);	*destreg* := *destreg* AND *constant* *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
and(*constant*, *destmem*);	*destmem* := *destmem* AND *constant* *destmem* may be any 8-bit, 16-bit, or 32-bit memory variable.

(continued)

Table A-5: HLA Syntax for and (continued)

Instruction	Description
and(*srcreg*, *destreg*);	*destreg* := *destreg* AND *srcreg* *destreg* and *srcreg* may be any 8-bit, 16-bit, or 32-bit general-purpose registers, though they must both be the same size.
and(*srcmem*, *destreg*);	*destreg* := *destreg* AND *srcmem* *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register; *srcmem* can be any like-sized memory location.
and(*srcreg*, *destmem*);	*destmem* := *destmem* AND *srcreg* *srcreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register; *destmem* can be any like-sized memory location.

Table A-6: Gas Syntax for and

Instruction	Description
andb *constant*, $destreg_8$ andw *constant*, $destreg_{16}$ andl *constant*, $destreg_{32}$	$destreg_n$:= $destreg_n$ AND *constant* $destreg_n$ may be any 8-bit, 16-bit, or 32-bit general-purpose register, as appropriate for the suffix.
andb *constant*, $destmem_8$ andw *constant*, $destmem_{16}$ andl *constant*, $destmem_{32}$	$destmem_n$:= $destmem_n$ AND *constant* $destmem_n$ may be any 8-bit, 16-bit, or 32-bit memory variable, as appropriate for the suffix.
andb $srcreg_8$, $destreg_8$ andw $srcreg_{16}$, $destreg_{16}$ andl $srcreg_{32}$, $destreg_{32}$ andq $srcreg_{64}$, $destreg_{64}$	$destreg_n$:= $destreg_n$ AND $srcreg_n$ $destreg_n$ and $srcreg_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose registers, as specified by the suffix.
andb $srcmem_8$, $destreg_8$ andw $srcmem_{16}$, $destreg_{16}$ andl $srcmem_{32}$, $destreg_{32}$ andq $srcmem_{64}$, $destreg_{64}$	$destreg_n$:= $destreg_n$ AND $srcmem_n$ $destreg_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, according to the suffix; $srcmem_n$ can be any like-sized memory location.
andb $srcreg_8$, $destmem_8$ andw $srcreg_{16}$, $destmem_{16}$ andl $srcreg_{32}$, $destmem_{32}$ andq $srcreg_{64}$, $destmem_{64}$	$destmem_n$:= $destmem_n$ AND $srcreg_n$ $srcreg_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, as specified by the suffix; $destmem_n$ can be any like-sized memory location.

Table A-7: MASM Syntax for and

Instruction	Description
and *destreg*, *constant*	*destreg* := *destreg* AND *constant* *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
and *destmem*, *constant*	*destmem* := *destmem* AND *constant* *destmem* may be any 8-bit, 16-bit, or 32-bit memory variable.

(continued)

Table A-7: MASM Syntax for and (continued)

Instruction	Description
and *destreg*, *srcreg*	*destreg* := *destreg* AND *srcreg* *destreg* and *srcreg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose registers, though they must both be the same size.
and *destreg*, *srcmem*	*destreg* := *destreg* AND *srcmem* *destreg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register; *srcmem* can be any like-sized memory location.
and *destmem*, *srcreg*	*destmem* := *destmem* AND *srcreg* *srcreg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register; *destmem* can be any like-sized memory location.

Table A-8: EFLAGS Settings for and

Flag	Setting
Carry	Always clear.
Overflow	Always clear.
Sign	Set if the result has a 1 in its HO bit position.
Zero	Set if the result is 0.

A.3 call

The call instruction requires a single operand. This instruction pushes the address of the instruction immediately following the call onto the 80x86 stack (see the discussion of the push instruction for a description of this operation). Then it transfers control to the address specified by the single operand and continues execution there. This instruction does not affect any flags.

Table A-9: HLA Syntax for call

Instruction	Description
call *label*; call(*label*);	Calls the subroutine that has the specified name (*label*) in the program.
call(reg_{32});	Calls the subroutine at the address specified in the 32-bit register supplied as the single operand (x86).
call(mem_{32});	Calls the subroutine at the address held in the double-word memory location specified by the mem_{32} operand.

Table A-10: Gas Syntax for `call`

Instruction	Description
`call` *label*	Calls the subroutine that has the specified name (*label*) in the program.
`call` *reg_{32}	Calls the subroutine at the address specified in the 32-bit register supplied as the single operand (x86).
`call` *reg_{64}	Calls the subroutine at the address specified in the 64-bit register supplied as the single operand (x86-64).
`call` *mem_{32}	Calls the subroutine at the address held in the double-word memory location specified by the mem_{32} operand (x86).
`call` *mem_{64}	Calls the subroutine at the address held in the quad-word memory location specified by the mem_{64} operand (x86-64).

Table A-11: MASM Syntax for `call`

Instruction	Description
`call` *label*	Calls the subroutine that has the specified name (*label*) in the program.
`call` reg_{32}	Calls the subroutine at the address specified in the 32-bit register supplied as the single operand (x86).
`call` reg_{64}	Calls the subroutine at the address specified in the 64-bit register supplied as the single operand (x86-64).
`call` mem_{32}	Calls the subroutine at the address held in the double-word memory location specified by the mem_{32} operand (x86).
`call` mem_{64}	Calls the subroutine at the address held in the quad-word memory location specified by the mem_{64} operand (x86-64).

A.4 clc, cmc, stc

The `clc` instruction clears the carry flag setting in the EFLAGS register. The `cmc` instruction complements (inverts) the carry flag. The `stc` instruction sets the carry flag. These instructions do not have any operands.

Table A-12: HLA Syntax for `clc`, `cmc`, and `stc`

Instruction	Description
`clc();`	Clears the carry flag
`cmc();`	Complements (inverts) the carry flag
`stc();`	Set the carry flag

Table A-13: Gas Syntax for `clc`, `cmc`, and `stc`

Instruction	Description
`clc`	Clears the carry flag
`cmc`	Complements (inverts) the carry flag
`stc`	Sets the carry flag

Table A-14: MASM Syntax for `clc`, `cmc`, and `stc`

Instruction	Description
`clc`	Clears the carry flag
`cmc`	Complements (inverts) the carry flag
`stc`	Sets the carry flag

A.5 cmp

The `cmp` instruction requires two operands: a left operand and a right operand. It compares the left operand to the right operand and sets the EFLAGS register based on the comparison. This instruction typically precedes a conditional jump instruction or some other instruction that tests the bits in the EFLAGS register.

Table A-15: HLA Syntax for `cmp`

Instruction	Description
`cmp(` *`reg`*`,` *`constant`* `);`	Compares *`reg`* against a *`constant`*. *`reg`* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
`cmp(` *`mem`*`,` *`constant`*`);`	Compares *`mem`* against a *`constant`*. *`destmem`* may be any 8-bit, 16-bit, or 32-bit memory variable.
`cmp(` *`leftreg`*`,` *`rightreg`* `);`	Compares *`leftreg`* against *`rightreg`*. *`leftreg`* and *`rightreg`* may be any 8-bit, 16-bit, or 32-bit general-purpose registers, though they must both be the same size.
`cmp(` *`reg`*`,` *`mem`*`);`	Compares a register with the value of a memory location. *`reg`* may be any 8-bit, 16-bit, or 32-bit general-purpose register; *`mem`* can be any like-sized memory location.
`cmp(` *`mem`*`,` *`reg`* `);`	Compares the value of a memory location against the value of a register. *`reg`* may be any 8-bit, 16-bit, or 32-bit general-purpose register; *`mem`* can be any like-sized memory location.

Table A-16: Gas Syntax for `cmp`

Instruction	Description
`cmpb` *constant*, reg_8 `cmpw` *constant*, reg_{16} `cmpl` *constant*, reg_{32}	Compares reg_n against a *constant*. reg_n may be any 8-bit, 16-bit, or 32-bit general-purpose register, as appropriate for the suffix.
`cmpb` *constant*, mem_8 `cmpw` *constant*, mem_{16} `cmpl` *constant*, mem_{32}	Compares mem_n against a *constant*. mem_n may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable, as appropriate for the suffix.
`cmpb` $leftreg_8$, $rightreg_8$ `cmpw` $leftreg_{16}$, $rightreg_{16}$ `cmpl` $leftreg_{32}$, $rightreg_{32}$ `cmpq` $leftreg_{64}$, $rightreg_{64}$	Compares $rightreg_n$ to $leftreg_n$ $rightreg_n$ and $leftreg_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose registers, as specified by the suffix.
`cmpb` mem_8, reg_8 `cmpw` mem_{16}, reg_{16} `cmpl` mem_{32}, reg_{32} `cmpq` mem_{64}, reg_{64}	Compares reg_n to mem_n reg_n may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, according to the suffix; mem_n can be any like-sized memory location.
`cmpb` reg_8, mem_8 `cmpw` reg_{16}, mem_{16} `cmpl` reg_{32}, mem_{32} `cmpl` reg_{64}, mem_{64}	Compares mem_n to reg_n reg_n may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, as specified by the suffix; mem_n can be any like-sized memory location.

Table A-17: MASM Syntax for `cmp`

Instruction	Description
`cmp` *reg*, *constant*	Compares *reg* against a *constant*. *reg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
`cmp` *mem*, *constant*	Compares *mem* against a *constant*. *destmem* may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable.
`cmp` *leftreg*, *rightreg*	Compares *leftreg* against *rightreg*. *leftreg* and *rightreg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose registers, though they must both be the same size.
`cmp` *reg*, *mem*	Compares a register with the value of a memory location. *reg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register; *mem* can be any like-sized memory location.
`cmp` *mem*, *reg*	Compares the value of a memory location against the value of a register. *reg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register; *mem* can be any like-sized memory location.

Table A-18: EFLAGS Settings for `cmp`

Flag	Setting
Carry	Set if the left (right for Gas) operand is less than the right (left for Gas) operand when performing an unsigned comparison.
Overflow/ sign	If the exclusive-OR of the overflow and sign flags is `1` after a comparison, then the first operand is less than the second operand in an unsigned comparison (for MASM and HLA; reverse the operands for Gas).
Zero	Set if the two values are equal.

A.6 dec

The dec (decrement) instruction requires a single operand. The CPU subtracts 1 from this operand. This instruction also sets several flags in the EFLAGS register, based on the result, but note that the flags are not set identically to the sub instruction.

Table A-19: HLA Syntax for dec

Instruction	Description
dec(*reg*);	*reg* := *reg* – 1 *reg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
dec(*mem*);	*mem* := *mem* – 1 *mem* may be any 8-bit, 16-bit, or 32-bit memory variable.

Table A-20: Gas Syntax for dec

Instruction	Description
decb reg_8 decw reg_{16} decl reg_{32} decq reg_{64}	reg_n := reg_n – 1 reg_n may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, as appropriate for the suffix.
decb mem_8 decw mem_{16} decl mem_{32} decq reg_{64}	mem_n := mem_n – 1 mem_n may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable, as appropriate for the suffix.

Table A-21: MASM Syntax for dec

Instruction	Description
dec *reg*	*reg* := *reg* – 1 *reg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register.
dec *mem*	*mem* := *mem* – 1 *mem* may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable.

Table A-22: EFLAGS Settings for dec

Flag	Setting
Carry	Unaffected by the dec instruction.
Overflow	Set if subtracting 1 produces a signed underflow.
Sign	Set if subtracting 1 produces a 1 in the HO bit position.
Zero	Set if subtracting 1 produces 0.

A.7 div

The div instruction takes a single operand. If that operand is an 8-bit operand (register or memory), div divides the 16-bit value in AX by that operand, producing the unsigned quotient in AL and the unsigned remainder in AH. If that operand is a 16-bit operand, div divides the 32-bit value in DX:AX (DX contains the HO word, and AX contains the LO word), leaving the unsigned quotient in AX and the unsigned remainder in DX. If the operand is a 32-bit operand, div divides the 64-bit quantity in EDX:EAX (EDX contains the HO double word, and EAX contains the LO double word) by the operand, leaving the unsigned quotient in EAX and the unsigned remainder in EDX. If the operand is a 64-bit operand, div divides the 128-bit quantity in RDX:RAX (RDX contains the HO quad word, and RAX contains the LO quad word) by the operand, leaving the unsigned quotient in RAX and the unsigned remainder in RDX.

This instruction scrambles the flags in the EFLAGS register; you cannot rely on flag values after executing a div. It also raises an integer divide exception if you attempt a division by zero or if the quotient will not fit in AL, AX, EAX, or RAX (as appropriate).

Table A-23: HLA Syntax for div

Instruction	Description
div(reg_8);	al := ax div reg_8 ah := ax mod reg_8 reg_8 may be any 8-bit general-purpose register.
div(reg_{16});	ax := dx:ax div reg_{16} dx := dx:ax mod reg_{16} reg_{16} may be any 16-bit general-purpose register.
div(reg_{32});	eax := edx:eax div reg_{32} edx := edx:eax mod reg_{32} reg_{32} may be any 32-bit general-purpose register.
div(mem_8);	al := ax div mem_8 ah := ax mod mem_8 mem_8 may be any 8-bit memory location.
div(mem_{16});	ax := dx:ax div mem_{16} dx := dx:ax mod mem_{16} mem_{16} may be any 16-bit memory location.
div(mem_{32});	eax := edx:eax div mem_{32} edx := edx:eax mod mem_{32} mem_{32} may be any 32-bit memory location.

Table A-24: Gas Syntax for `div`

Instruction	Description
`divb` *reg*$_8$	`al := ax div` *reg*$_8$ `ah := ax mod` *reg*$_8$ *reg*$_8$ may be any 8-bit general-purpose register.
`divw` *reg*$_{16}$	`ax := dx:ax div` *reg*$_{16}$ `dx := dx:ax mod` *reg*$_{16}$ *reg*$_{16}$ may be any 16-bit general-purpose register.
`divl` *reg*$_{32}$	`eax := edx:eax div` *reg*$_{32}$ `edx := edx:eax mod` *reg*$_{32}$ *reg*$_{32}$ may be any 32-bit general-purpose register.
`divq` *reg*$_{64}$	`rax := rdx:rax div` *reg*$_{64}$ `rdx := rdx:rax mod` *reg*$_{64}$ *reg*$_{64}$ may be any 64-bit general-purpose register.
`divb` *mem*$_8$	`al := ax div` *mem*$_8$ `ah := ax mod` *mem*$_8$ *mem*$_8$ may be any 8-bit memory location.
`divw` *mem*$_{16}$	`ax := dx:ax div` *mem*$_{16}$ `dx := dx:ax mod` *mem*$_{16}$ *mem*$_{16}$ may be any 16-bit memory location.
`divl` *mem*$_{32}$	`eax := edx:eax div` *mem*$_{32}$ `edx := edx:eax mod` *mem*$_{32}$ *mem*$_{32}$ may be any 32-bit memory location.
`divq` *mem*$_{64}$	`rax := rdx:rax div` *mem*$_{64}$ `rdx := rdx:rax mod` *mem*$_{64}$ *mem*$_{64}$ may be any 64-bit memory location.

Table A-25: MASM Syntax for `div`

Instruction	Description
`div` *reg*$_8$	`al := ax div` *reg*$_8$ `ah := ax mod` *reg*$_8$ *reg*$_8$ may be any 8-bit general-purpose register.
`div` *reg*$_{16}$	`ax := dx:ax div` *reg*$_{16}$ `dx := dx:ax mod` *reg*$_{16}$ *reg*$_{16}$ may be any 16-bit general-purpose register.
`div` *reg*$_{32}$	`eax := edx:eax div` *reg*$_{32}$ `edx := edx:eax mod` *reg*$_{32}$ *reg*$_{32}$ may be any 32-bit general-purpose register.
`div` *reg*$_{64}$	`rax := rdx:rax div` *reg*$_{64}$ `rdx := rdx:rax mod` *reg*$_{64}$ *reg*$_{64}$ may be any 64-bit general-purpose register.
`div` *mem*$_8$	`al := ax div` *mem*$_8$ `ah := ax mod` *mem*$_8$ *mem*$_8$ may be any 8-bit memory location.

(continued)

Table A-25: MASM Syntax for `div` (continued)

Instruction	Description
`div` *mem*$_{16}$	`ax := dx:ax div` *mem*$_{16}$ `dx := dx:ax mod` *mem*$_{16}$ *mem*$_{16}$ may be any 16-bit memory location.
`div` *mem*$_{32}$	`eax := edx:eax div` *mem*$_{32}$ `edx := edx:eax mod` *mem*$_{32}$ *mem*$_{32}$ may be any 32-bit memory location.
`div` *mem*$_{64}$	`rax := rdx:rax div` *mem*$_{64}$ `rdx := rdx:rax mod` *mem*$_{64}$ *mem*$_{64}$ may be any 64-bit memory location.

Table A-26: EFLAGS Settings for `div`

Flag	Setting
Carry	Scrambled by the `div` instruction.
Overflow	Scrambled by the `div` instruction.
Sign	Scrambled by the `div` instruction.
Zero	Scrambled by the `div` instruction.

A.8 enter

The `enter` instruction completes construction of an *activation record* (see "Activation Records and the Stack" on page 563) upon entry into a procedure. This instruction pushes the value of EBP (RBP on x86-64) onto the stack (see the discussion of the `push` instruction, later in this appendix, for details). It subtracts the value of its *`locals`* argument from the ESP/RSP register to allocate storage for local variables. If the *`lexlevel`* argument is nonzero, the `enter` instruction builds a *display*. (You won't encounter displays very often, so they're not discussed in this book. For more details, see *The Art of Assembly Language*, 2nd ed. [No Starch Press, 2010].) Most compilers, when they even use the `enter` instruction, specify `0` as the *`lexlevel`* operand.

Table A-27: HLA Syntax for `enter`

Instruction	Description

`enter(` *locals*$_{16}$`,` *lexlevel*$_{8}$ `);`	`push( ebp );`
	`sub(` *locals*$_{16}$`, ESP );`
	Build display if *lexlevel*$_{8}$ is nonzero.
	Note: *locals*$_{16}$ is a 16-bit constant; *lexlevel*$_{8}$ is an 8-bit constant.

Table A-28: Gas Syntax for `enter`

Instruction	Description

`enter` *lexlevel*$_8$, *locals*$_{16}$	`push( ebp );`
	`sub(` *locals*$_{16}$`, esp); // RSP on x86-64`
	Build display if *lexlevel*$_8$ is nonzero.
	Note: *locals*$_{16}$ is a 16-bit constant; *lexlevel*$_8$ is an 8-bit constant.

Table A-29: MASM Syntax for `enter`

Instruction	Description

`enter` *locals*$_{16}$, *lexlevel*$_8$	`push( ebp );`
	`sub(` *locals*$_{16}$`, esp); // RSP on x86-64`
	Build display if *lexlevel*$_8$ is nonzero.
	Note: *locals*$_{16}$ is a 16-bit constant; *lexlevel*$_8$ is an 8-bit constant.

Table A-30: EFLAGS Settings for `enter`

Flag	Setting
Carry	Unaffected by the `enter` instruction.
Overflow	Unaffected by the `enter` instruction.
Sign	Unaffected by the `enter` instruction.
Zero	Unaffected by the `enter` instruction.

A.9 idiv

The `idiv` instruction takes a single operand. If that operand is an 8-bit operand (register or memory), `idiv` divides the 16-bit value in AX by that operand, producing the signed quotient in AL and the signed remainder in AH. If that operand is a 16-bit operand, `idiv` divides the 32-bit value in DX:AX (DX contains the HO word, and AX contains the LO word), leaving the signed quotient in AX and the signed remainder in DX. If the operand is a 32-bit operand, `idiv` divides the 64-bit quantity in EDX:EAX (EDX contains the HO double word, and EAX contains the LO double word) by the operand, leaving the signed quotient in EAX and the signed remainder in EDX. If the operand is a 64-bit operand, `idiv` divides the 128-bit quantity in RDX:RAX (RDX contains the HO quad word, and RAX contains the LO quad word) by the operand, leaving the signed quotient in RAX and the signed remainder in RDX.

This instruction scrambles the flags in the EFLAGS register; you cannot rely on flag values after executing an `idiv` instruction. This instruction raises an integer divide exception if you attempt a division by zero or if the quotient will not fit in AL, AX, EAX, or RAX (as appropriate).

Table A-31: HLA Syntax for `idiv`

Instruction	Description
`idiv(` *reg*$_8$ `);`	`al := ax div` *reg*$_8$ `ah := ax mod` *reg*$_8$ *reg*$_8$ may be any 8-bit general-purpose register.
`idiv(` *reg*$_{16}$ `);`	`ax := dx:ax div` *reg*$_{16}$ `dx := dx:ax mod` *reg*$_{16}$ *reg*$_{16}$ may be any 16-bit general-purpose register.
`idiv(` *reg*$_{32}$ `);`	`eax := edx:eax div` *reg*$_{32}$ `edx := edx:eax mod` *reg*$_{32}$ *reg*$_{32}$ may be any 32-bit general-purpose register.
`idiv(` *mem*$_8$ `);`	`al := ax div` *mem*$_8$ `ah := ax mod` *mem*$_8$ *mem*$_8$ may be any 8-bit memory location.
`idiv(` *mem*$_{16}$ `);`	`ax := dx:ax div` *mem*$_{16}$ `dx := dx:ax mod` *mem*$_{16}$ *mem*$_{16}$ may be any 16-bit memory location.
`idiv(` *mem*$_{32}$ `);`	`eax := edx:eax div` *mem*$_{32}$ `edx := edx:eax mod` *mem*$_{32}$ *mem*$_{32}$ may be any 32-bit memory location.

Table A-32: Gas Syntax for `idiv`

Instruction	Description
`idivb` *reg*$_8$	`al := ax div` *reg*$_8$ `ah := ax mod` *reg*$_8$ *reg*$_8$ may be any 8-bit general-purpose register.
`idivw` *reg*$_{16}$	`ax := dx:ax div` *reg*$_{16}$ `dx := dx:ax mod` *reg*$_{16}$ *reg*$_{16}$ may be any 16-bit general-purpose register.
`idivl` *reg*$_{32}$	`eax := edx:eax div` *reg*$_{32}$ `edx := edx:eax mod` *reg*$_{32}$ *reg*$_{32}$ may be any 32-bit general-purpose register.
`idivq` *reg*$_{32}$	`rax := rdx:rax div` *reg*$_{64}$ `rdx := rdx:rax mod` *reg*$_{64}$ *reg*$_{64}$ may be any 64-bit general-purpose register.
`idivb` *mem*$_8$	`al := ax div` *mem*$_8$ `ah := ax mod` *mem*$_8$ *mem*$_8$ may be any 8-bit memory location.
`idivw` *mem*$_{16}$	`ax := dx:ax div` *mem*$_{16}$ `dx := dx:ax mod` *mem*$_{16}$ *mem*$_{16}$ may be any 16-bit memory location.
`idivl` *mem*$_{32}$	`eax := edx:eax div` *mem*$_{32}$ `edx := edx:eax mod` *mem*$_{32}$ *mem*$_{32}$ may be any 32-bit memory location.
`idivl` *mem*$_{64}$	`rax := rdx:rax div` *mem*$_{64}$ `rdx := rdx:rax mod` *mem*$_{64}$ *mem*$_{64}$ may be any 64-bit memory location.

Table A-33: MASM Syntax for `idiv`

Instruction	Description
`idiv` *reg*$_8$	`al := ax div` *reg*$_8$ `ah := ax mod` *reg*$_8$ *reg*$_8$ may be any 8-bit general-purpose register.
`idiv` *reg*$_{16}$	`ax := dx:ax div` *reg*$_{16}$ `dx := dx:ax mod` *reg*$_{16}$ *reg*$_{16}$ may be any 16-bit general-purpose register.
`idiv` *reg*$_{32}$	`eax := edx:eax div` *reg*$_{32}$ `edx := edx:eax mod` *reg*$_{32}$ *reg*$_{32}$ may be any 32-bit general-purpose register.
`idiv` *reg*$_{64}$	`rax := rdx:rax div` *reg*$_{64}$ `rdx := rdx:rax mod` *reg*$_{64}$ *reg*$_{64}$ may be any 64-bit general-purpose register.
`idiv` *mem*$_8$	`al := ax div` *mem*$_8$ `ah := ax mod` *mem*$_8$ *mem*$_8$ may be any 8-bit memory location.
`idiv` *mem*$_{16}$	`ax := dx:ax div` *mem*$_{16}$ `dx := dx:ax mod` *mem*$_{16}$ *mem*$_{16}$ may be any 16-bit memory location.
`idiv` *mem*$_{32}$	`eax := edx:eax div` *mem*$_{32}$ `edx := edx:eax mod` *mem*$_{32}$ *mem*$_{32}$ may be any 32-bit memory location.
`idiv` *mem*$_{64}$	`rax := rdx:rax div` *mem*$_{64}$ `rdx := rdx:rax mod` *mem*$_{64}$ *mem*$_{64}$ may be any 64-bit memory location.

Table A-34: EFLAGS Settings for `idiv`

Flag	Setting
Carry	Scrambled by the `idiv` instruction.
Overflow	Scrambled by the `idiv` instruction.
Sign	Scrambled by the `idiv` instruction.
Zero	Scrambled by the `idiv` instruction.

A.10 imul, intmul

The `imul` instruction takes a couple of forms. In HLA, MASM, and Gas, one form of this instruction has a single operand. If that operand is an 8-bit operand (register or memory), `imul` multiplies the 8-bit value in AL by that operand, producing the signed product in AX. If the operand is 16 bits, `imul` multiplies the 16-bit value in AX by the operand, leaving the signed product in DX:AX (DX contains the HO word, and AX contains the LO word). If the operand is 32 bits, `imul` multiples the 32-bit quantity in EAX by the operand, leaving the signed product in EDX:EAX (EDX

contains the HO double word, and EAX contains the LO double word). If the operand is a 64-bit operand, `imul` multiples the 64-bit quantity in RAX by the operand, leaving the signed product in RDX:RAX (RDX contains the HO quad word, and RAX contains the LO quad word).

This instruction scrambles the zero and sign flags in the EFLAGS register; you cannot rely on flag values after executing an `imul` instruction. It sets the carry and overflow flags if the result doesn't fit into the size specified by the single operand.

A second form of the integer multiply instruction exists that does not produce an extended-precision result. Gas and MASM continue to use the `imul` mnemonic for this instruction, while HLA uses the `intmul` mnemonic (because the semantics are different for this instruction, it deserves a different mnemonic).

Table A-35: HLA Syntax for `imul`

Instruction	Description
`imul(` *reg*$_8$ `);`	`ax := al *` *reg*$_8$ *reg*$_8$ may be any 8-bit general-purpose register.
`imul(` *reg*$_{16}$ `);`	`dx:ax := ax *` *reg*$_{16}$ *reg*$_{16}$ may be any 16-bit general-purpose register.
`imul(` *reg*$_{32}$ `);`	`edx:eax := eax *` *reg*$_{32}$ *reg*$_{32}$ may be any 32-bit general-purpose register.
`imul(` *mem*$_8$ `);`	`ax := al *` *mem*$_8$ *mem*$_8$ may be any 8-bit memory location.
`imul(` *mem*$_{16}$ `);`	`dx:ax := ax *` *mem*$_{16}$ *mem*$_{16}$ may be any 16-bit memory location.
`imul(` *mem*$_{32}$ `);`	`edx:eax := eax *` *mem*$_{32}$ *mem*$_{32}$ may be any 32-bit memory location.
`intmul(` *constant*`,` *srcreg*`,` *destreg*`);`	*destreg* `:=` *srcreg* `*` *constant* *srcreg* and *destreg* may be 16-bit or 32-bit general-purpose registers; they must both be the same size.
`intmul(` *constant*`,` *destreg*`);`	*destreg* `:=` *destreg* `*` *constant* *destreg* may be a 16-bit or 32-bit general-purpose register.
`intmul(` *srcreg*`,` *destreg*`);`	*destreg* `:=` *srcreg* `*` *destreg* *srcreg* and *destreg* may be 16-bit or 32-bit general-purpose registers; they must both be the same size.
`intmul(` *mem*`,` *destreg*`);`	*destreg* `:=` *mem* `*` *destreg* *destreg* must be a 16-bit or 32-bit general-purpose register; *mem* is a memory location that must be the same size as the register.

Table A-36: Gas Syntax for `imul`

Instruction	Description
`imulb` reg_8	`ax := al *` reg_8 reg_8 may be any 8-bit general-purpose register.
`imulw` reg_{16}	`dx:ax := ax *` reg_{16} reg_{16} may be any 16-bit general-purpose register.
`imull` reg_{32}	`edx:eax := eax *` reg_{32} reg_{32} may be any 32-bit general-purpose register.
`imulq` reg_{32}	`rdx:rax := rax *` reg_{64} reg_{64} may be any 64-bit general-purpose register.
`imulb` mem_8	`ax := al *` mem_8 mem_8 may be any 8-bit memory location.
`imulw` mem_{16}	`dx:ax := ax *` mem_{16} mem_{16} may be any 16-bit memory location.
`imull` mem_{32}	`edx:eax := eax *` mem_{32} mem_{32} may be any 32-bit memory location.
`imull` mem_{64}	`rdx:rax := rax *` mem_{64} mem_{64} may be any 64-bit memory location.
`imulw` *constant*, *srcreg*, *destreg*	*destreg* := *srcreg* * *constant* *srcreg* and *destreg* must be 16-bit general-purpose registers.
`imull` *constant*, *srcreg*, *destreg*	*destreg* := *srcreg* * *constant* *srcreg* and *destreg* must be 32-bit general-purpose registers.
`imulw` *constant*, *destreg*	*destreg* := *destreg* * *constant* *destreg* must be a 16-bit general-purpose register.
`imull` *constant*, *destreg*	*destreg* := *destreg* * *constant* *destreg* may be a 32-bit general-purpose register.
`imulw` *srcreg*, *destreg*	*destreg* := *srcreg* * *destreg* *srcreg* and *destreg* must both be 16-bit general-purpose registers.
`imull` *srcreg*, *destreg*	*destreg* := *srcreg* * *destreg* *srcreg* and *destreg* must both be 32-bit general-purpose registers.
`imulq` *srcreg*, *destreg*	*destreg* := *srcreg* * *destreg* *srcreg* and *destreg* must both be 64-bit general-purpose registers.
`imulw` *mem*, *destreg*	*destreg* := *mem* * *destreg* *destreg* must be a 16-bit general-purpose register; *mem* is a memory location that must be the same size as the register.
`imull` *mem*, *destreg*	*destreg* := *mem* * *destreg* *destreg* must be a 32-bit general-purpose register; *mem* is a memory location that must be the same size as the register.
`imulq` *mem*, *destreg*	*destreg* := *mem* * *destreg* *destreg* must be a 64-bit general-purpose register; *mem* is a memory location that must be the same size as the register.

Table A-37: MASM Syntax for `imul`

Instruction	Description
`imul` reg_8	`ax := al *` reg_8 reg_8 may be any 8-bit general-purpose register.
`imul` reg_{16}	`dx:ax := ax *` reg_{16} reg_{16} may be any 16-bit general-purpose register.
`imul` reg_{32}	`edx:eax := eax *` reg_{32} reg_{32} may be any 32-bit general-purpose register.
`imul` reg_{64}	`rdx:rax := rax *` reg_{64} reg_{64} may be any 64-bit general-purpose register.
`imul` mem_8	`ax := al *` mem_8 mem_8 may be any 8-bit memory location.
`imul` mem_{16}	`dx:ax := ax *` mem_{16} mem_{16} may be any 16-bit memory location.
`imul` mem_{32}	`edx:eax := eax *` mem_{32} mem_{32} may be any 32-bit memory location.
`imul` mem_{64}	`rdx:rax := rax *` mem_{64} mem_{64} may be any 64-bit memory location.
`imul` *destreg*, *srcreg*, *constant*	*destreg* := *srcreg* * *constant* *srcreg* and *destreg* may be 16-bit or 32-bit general-purpose registers; they must both be the same size.
`imul` *destreg*, *constant*	*destreg* := *destreg* * *constant* *destreg* may be a 16-bit or 32-bit general-purpose register.
`imul` *destreg*, *srcreg*	*destreg* := *srcreg* * *destreg* *srcreg* and *destreg* may be 16-bit, 32-bit, or 64-bit general-purpose registers; they must both be the same size.
`imul` *destreg*, *mem*	*destreg* := *mem* * *destreg* *destreg* must be a 16-bit, 32-bit, or 64-bit general-purpose register; *mem* is a memory location that must be the same size as the register.

Table A-38: EFLAGS Settings for `imul`

Flag	Setting
Carry	Set if signed overflow occurs.
Overflow	Set if signed overflow occurs.
Sign	Scrambled by the `idiv` instruction.
Zero	Scrambled by the `idiv` instruction.

A.11 inc

The inc (increment) instruction requires a single operand. The CPU adds 1 from this operand. This instruction also sets several flags in the EFLAGS register, based on the result, but note that the flags are not set identically to the add instruction.

Table A-39: HLA Syntax for inc

Instruction	Description
inc(*reg*);	*reg* := *reg* + 1 *reg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
inc(*mem*);	*mem* := *mem* + 1 *mem* may be any 8-bit, 16-bit, or 32-bit memory variable.

Table A-40: Gas Syntax for inc

Instruction	Description
incb reg_8 incw reg_{16} incl reg_{32} incq reg_{64}	reg_n := reg_n + 1 reg_n may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, as appropriate for the suffix.
incb mem_8 incw mem_{16} incl mem_{32} incq mem_{64}	mem_n := mem_n + 1 mem_n may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable, as appropriate for the suffix.

Table A-41: MASM Syntax for inc

Instruction	Description
inc *reg*	*reg* := *reg* + 1 *reg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register.
inc *mem*	*mem* := *mem* + 1 *mem* may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable.

Table A-42: EFLAGS Settings for inc

Flag	Setting
Carry	Unaffected by the inc instruction.
Overflow	Set if adding 1 produces a signed overflow.
Sign	Set if adding 1 produces a 1 in the HO bit position.
Zero	Set if adding 1 produces 0.

A.12 Conditional Jumps

The 80x86 supports a wide variety of conditional jumps (`jcc`) that allow the CPU to make decisions based on conditions computed by instructions, such as `cmp`, that affect various flags in the EFLAGS register. Note, however, that these instructions do not themselves affect any of the flags in the EFLAGS register.

Table A-43: Conditional Jump Instructions

Instruction	Description
`ja` *`label`*`;` `//HLA` `ja` *`label`* `//Gas/MASM`	Unsigned conditional jump if above. You would generally use this instruction immediately after a `cmp` instruction to test to see if one operand is greater than another using an unsigned comparison. Control transfers to the specified *`label`* if this condition is `true`, and falls through to the next instruction if the condition is `false`.
`jae` *`label`*`;` `//HLA` `jae` *`label`* `//Gas/MASM`	Unsigned conditional jump if above or equal. See `ja` for details.
`jb` *`label`*`;` `//HLA` `jb` *`label`* `//Gas/MASM`	Unsigned conditional jump if below. See `ja` for details.
`jbe` *`label`*`;` `//HLA` `jbe` *`label`* `//Gas/MASM`	Unsigned conditional jump if below or equal. See `ja` for details.
`jc` *`label`*`;` `//HLA` `jc` *`label`* `//Gas/MASM`	Conditional jump if carry is `1`. See `ja` for details.
`je` *`label`*`;` `//HLA` `je` *`label`* `//Gas/MASM`	Conditional jump if equal. See `ja` for details.
`jg` *`label`*`;` `//HLA` `jg` *`label`* `//Gas/MASM`	Signed conditional jump if greater. See `ja` for details.
`jge` *`label`*`;` `//HLA` `jge` *`label`* `//Gas/MASM`	Signed conditional jump if greater or equal. See `ja` for details.
`jl` *`label`*`;` `//HLA` `jl` *`label`* `//Gas/MASM`	Signed conditional jump if less than. See `ja` for details.
`jle` *`label`*`;` `//HLA` `jle` *`label`* `//Gas/MASM`	Signed conditional jump if less than or equal. See `ja` for details.
`jna` *`label`*`;` `//HLA` `jna` *`label`* `//Gas/MASM`	Unsigned conditional jump if not above. See `ja` for details.
`jnae` *`label`*`;` `//HLA` `jnae` *`label`* `//Gas/MASM`	Unsigned conditional jump if not above or equal. See `ja` for details.
`jnb` *`label`*`;` `//HLA` `jnb` *`label`* `//Gas/MASM`	Unsigned conditional jump if below. See `ja` for details.
`jnbe` *`label`*`;` `//HLA` `jnbe` *`label`* `//Gas/MASM`	Unsigned conditional jump if below or equal. See `ja` for details.
`jnc` *`label`*`;` `//HLA` `jnc` *`label`* `//Gas/MASM`	Conditional jump if carry flag is clear (no carry). See `ja` for details.

(continued)

Table A-43: Conditional Jump Instructions (continued)

Instruction		Description
`jne` *label*;	`//HLA`	Conditional jump if not equal. See `ja` for details.
`jne` *label*	`//Gas/MASM`	
`jng` *label*;	`//HLA`	Signed conditional jump if not greater. See `ja` for details.
`jng` *label*	`//Gas/MASM`	
`jnge` *label*;	`//HLA`	Signed conditional jump if not greater or equal. See `ja` for details.
`jnge` *label*	`//Gas/MASM`	
`jnl` *label*;	`//HLA`	Signed conditional jump if not less than. See `ja` for details.
`jnl` *label*	`//Gas/MASM`	
`jnle` *label*;	`//HLA`	Signed conditional jump if not less than or equal. See `ja` for details.
`jnle` *label*	`//Gas/MASM`	
`jno` *label*;	`//HLA`	Conditional jump if no overflow (overflow flag = 0). See `ja` for details.
`jno` *label*	`//Gas/MASM`	
`jns` *label*;	`//HLA`	Conditional jump if no sign (sign flag = 0). See `ja` for details.
`jns` *label*	`//Gas/MASM`	
`jnz` *label*;	`//HLA`	Conditional jump if not zero (zero flag = 0). See `ja` for details.
`jnz` *label*	`//Gas/MASM`	
`jo` *label*;	`//HLA`	Conditional jump if overflow (overflow flag = 1). See `ja` for details.
`jo` *label*	`//Gas/MASM`	
`js` *label*;	`//HLA`	Conditional jump if sign (sign flag = 0). See `ja` for details.
`js` *label*	`//Gas/MASM`	
`jz` *label*;	`//HLA`	Conditional jump if zero (zero flag = 0). See `ja` for details.
`jz` *label*	`//Gas/MASM`	
`jcxz` *label*;	`//HLA`	Conditional jump if CX is 0. See `ja` for details. Note: the range of this branch is limited to ±128 bytes around the instruction.
`jcxz` *label*	`//Gas/MASM`	
`jecxz` *label*;	`//HLA`	Conditional jump if ECX is 0. See `ja` for details. Note: the range of this branch is limited to ±128 bytes around the instruction.
`jecxz` *label*	`//Gas/MASM`	

A.13 jmp

The `jmp` instruction unconditionally transfers control to the memory location specified by its operand. This instruction does not affect any flags.

Table A-44: HLA Syntax for `jmp`

Instruction	Description
`jmp` *label*; `jmp(` *label* `);`	Transfers control to the machine instruction following the *label* in the source file.
`jmp(` reg_{32} `);`	Transfers control to the memory location whose address is held in the 32-bit general-purpose register reg_{32}.
`jmp(` mem_{32} `);`	Transfers control to the memory location whose 32-bit address is held in the memory location specified by mem_{32}.

Table A-45: Gas Syntax for `jmp`

Instruction	Description
`jmp label`	Transfers control to the machine instruction following the `label` in the source file.
`jmp *`reg_{32} `jmp *`reg_{64}	Transfers control to the memory location whose address is held in the general-purpose register reg_{32} (x86) or reg_{64} (x86-64).
`jmp` mem_{32} `jmp` mem_{64}	Transfers control to the memory location whose address is held in the memory location specified by mem_{32} (x86) or mem_{64} (x86-64).

Table A-46: MASM Syntax for `jmp`

Instruction	Description
`jmp label`	Transfers control to the machine instruction following the `label` in the source file.
`jmp` reg_{32} `jmp` reg_{64}	Transfers control to the memory location whose address is held in the 32-bit general-purpose register reg_{32} (x86) or 64-bit general-purpose register reg_{64} (x86-64).
`jmp` mem_{32} `jmp` mem_{64}	Transfers control to the memory location whose address is held in the memory location specified by mem_{32} (x86) or mem_{64} (x86-64).

A.14 lea

The `lea` instruction loads a register with the effective address of a memory operand. This is in direct contrast to the `mov` instruction, which loads a register with the contents of a memory location. Like `mov`, the `lea` instruction does not affect any flags in the EFLAGS register. Many compilers actually use this instruction to add a constant to a register, or multiply a register's value by 2, 4, or 8, and then move the result into a different register.

Table A-47: HLA Syntax for `lea`

Instruction	Description
`lea(` reg_{32}`,` *mem* `);` `lea(` *mem*`,` reg_{32} `);`	reg_{32} := address of *mem* reg_{32} must be a 32-bit general-purpose register. *mem* can be any sized memory location. Note that both syntaxes are identical in HLA.

Table A-48: Gas Syntax for `lea`

Instruction	Description
`leal` *mem*`,` reg_{32} `leaq` *mem*`,` reg_{32}	reg_{32} := address of *mem* reg_{64} must be a 64-bit general-purpose register. *mem* can be any sized memory location.

Table A-49: MASM Syntax for `lea`

Instruction	Description
`lea` *reg_{32}*`,` *mem* `lea` *reg_{64}*`,` *mem*	*reg_{xx}* := address of *mem*. *reg_{32}* must be a 32-bit general-purpose register. *mem* can be any sized memory location (x86). *reg_{64}* must be a 64-bit general-purpose register. *mem* can be any sized memory location (x86-64).

A.15 leave

The `leave` instruction cleans up after a procedure, removing local variable storage and restoring the EBP (RPB on x86-64) register to its original value. It copies the value of EBP/RBP into ESP/RSP. It pops EBP/RBP's value from the stack.

Table A-50: HLA Syntax for `leave`

Instruction	Description

`leave; //HLA syntax`
`leave  //MASM/Gas`
	`mov( ebp, esp );`
	`pop( esp );`

Table A-51: EFLAGS Settings for `leave`

Flag	Setting
Carry	Unaffected by the `leave` instruction.
Overflow	Unaffected by the `leave` instruction.
Sign	Unaffected by the `leave` instruction.
Zero	Unaffected by the `leave` instruction.

A.16 mov

The `mov` instruction requires two operands: a source and a destination. It copies the value from the source operand to the destination operand. It does not affect any flags in the EFLAGS register.

Table A-52: HLA Syntax for `mov`

Instruction	Description
`mov(` *constant*`,` *destreg* `);`	*destreg* := *constant* *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
`mov(` *constant*`,` *destmem* `);`	*destmem* := *constant* *destmem* may be any 8-bit, 16-bit, or 32-bit memory variable.

(continued)

Table A-52: HLA Syntax for `mov` (continued)

Instruction	Description
`mov( srcreg, destreg );`	*destreg* := *srcreg* *destreg* and *srcreg* may be any 8-bit, 16-bit, or 32-bit general-purpose registers, though they must both be the same size.
`mov( srcmem, destreg );`	*destreg* := *srcmem* *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register; *srcmem* can be any like-sized memory location.
`mov( srcreg, destmem );`	*destmem* := *srcreg* *srcreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register; *destmem* can be any like-sized memory location.

Table A-53: Gas Syntax for `mov`

Instruction	Description
`movb` *constant*, *destreg*$_8$ `movw` *constant*, *destreg*$_{16}$ `movl` *constant*, *destreg*$_{32}$	*destreg*$_n$:= *constant* *destreg*$_n$ may be any 8-bit, 16-bit, or 32-bit general-purpose register, as appropriate for the suffix.
`movb` *constant*, *destmem*$_8$ `movw` *constant*, *destmem*$_{16}$ `movl` *constant*, *destmem*$_{32}$	*destmem*$_n$:= *constant* *destmem*$_n$ may be any 8-bit, 16-bit, or 32-bit memory variable, as appropriate for the suffix.
`movb` *srcreg*$_8$, *destreg*$_8$ `movw` *srcreg*$_{16}$, *destreg*$_{16}$ `movl` *srcreg*$_{32}$, *destreg*$_{32}$ `movq` *srcreg*$_{64}$, *destreg*$_{64}$	*destreg*$_n$:= *srcreg*$_n$ *destreg*$_n$ and *srcreg*$_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose registers, as specified by the suffix.
`movb` *srcmem*$_8$, *destreg*$_8$ `movw` *srcmem*$_{16}$, *destreg*$_{16}$ `movl` *srcmem*$_{32}$, *destreg*$_{32}$ `movq` *srcmem*$_{64}$, *destreg*$_{64}$	*destreg*$_n$:= *srcmem*$_n$ *destreg*$_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, according to the suffix; *srcmem*$_n$ can be any like-sized memory location.
`movb` *srcreg*$_8$, *destmem*$_8$ `movw` *srcreg*$_{16}$, *destmem*$_{16}$ `movl` *srcreg*$_{32}$, *destmem*$_{32}$ `movl` *srcreg*$_{64}$, *destmem*$_{64}$	*destmem*$_n$:= *srcreg*$_n$ *srcreg*$_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, as specified by the suffix; *destmem*$_n$ can be any like-sized memory location.

Table A-54: MASM Syntax for `mov`

Instruction	Description
`mov` *destreg*, *constant*	*destreg* := *constant* *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
`mov` *destmem*, *constant*	*destmem* := *constant* *destmem* may be any 8-bit, 16-bit, or 32-bit memory variable.
`mov` *destreg*, *srcreg*	*destreg* := *srcreg* *destreg* and *srcreg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose registers, though they must both be the same size.

(continued)

Table A-54: MASM Syntax for `mov` (continued)

Instruction	Description
`mov` *`destreg, srcmem`*	*`destreg`* `:=` *`srcmem`* *`destreg`* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register; *`srcmem`* can be any like-sized memory location.
`mov` *`destmem, srcreg`*	*`destmem`* `:=` *`srcreg`* *`srcreg`* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register; *`destmem`* can be any like-sized memory location.

A.17 movs, movsb, movsd, movsq, movsw

The `movs` instructions—the *move string* instructions that copy blocks of data from one range of memory locations to another—do not require any explicit operands. These instructions take two forms: the move string instruction by itself, and a move string instruction with a "repeat" prefix.

Without a repeat prefix, these instructions copy a byte (`movsb`), word (`movsw`), double word (`movsd`), or quad word (`movsq`) from the memory location pointed at by ESI (the source index register) to the memory location pointed at by EDI (the destination index register). After copying the data, the CPU either increments or decrements these two registers by the size, in bytes, of the transfer. That is, `movsb` increments or decrements ESI and EDI by 1, `movsw` increments or decrements them by 2, and `movsd` increments or decrements them by 4. These instructions determine whether to increment or decrement ESI and EDI based on the value of the *direction flag* in the EFLAGs register. If the direction flag is clear, the move string instructions increment ESI and EDI; if the direction flag is set, the move string instructions decrement ESI and EDI.

If the repeat prefix is attached to one of these move string instructions, the CPU repeats the move operation the number of times specified by the ECX register.

These instructions do not affect any flags.

Table A-55: HLA Syntax for `movsb`, `movsd`, and `movsw`

Instruction	Description

`movsb();`
`movsw();`
`movsd();`
	`[edi] := [esi]`
	Copies the byte, word, or double word pointed at by ESI to the memory location pointed at by EDI. After moving the data, these instructions increment ESI and EDI by 1, 2, or 4 if the direction flag is clear; they decrement ESI and EDI by 1, 2, or 4 if the direction flag is set.

`rep.movsb();`
`rep.movsw();`
`rep.movsd();`
	`[edi] := [esi]`
	Copies a block of ECX bytes, words, or double words from where ESI points to where EDI points. Increments or decrements ESI and EDI after each movement by the size of the data moved, based on the value of the direction flag.

Table A-56: Gas Syntax for `movsb`, `movsd`, `movsl`, and `movsw`

Instruction	Description

`movsb`
`movsw`
`movsl`
`movsq`
	`[edi] := [esi]  // x86`
	`[rdi] := [rsi]  // x86-64`
	Copies the byte, word, or double word pointed at by ESI (x86) or RSI (x86-64) to the memory location pointed at by EDI (x86) or RDI (x86-64). After moving the data, these instructions increment ESI/RSI and EDI/RDI by 1, 2, 4, or 8 if the direction flag is clear; they decrement ESI/RSI and EDI/RDI by 1, 2, 4, or 8 if the direction flag is set.

`rep movsb`
`rep movsw`
`rep movsl`
`rep movsq`
	`[edi] := [esi]  // x86 * count in ECX`
	`[rdi] := [rsi]  // x86-64 * count in RCX`
	Copies the byte, word, or double word pointed at by ESI (x86) or RSI (x86-64) to the memory location pointed at by EDI (x86) or RDI (x86-64). After moving the data, these instructions increment ESI/RSI and EDI/RDI by 1, 2, 4, or 8 if the direction flag is clear; they decrement ESI/RSI and EDI/RDI by 1, 2, 4, or 8 if the direction flag is set. These instructions repeat this operation the number of times specified by the count in the ECX (x86)/RCX (x86-64) register.

Table A-57: MASM Syntax for `movsb`, `movsd`, `movsq`, and `movsw`

Instruction	Description

`movsb`
`movsw`
`movsd`
`movsq`
	`[edi] := [esi]  // x86`
	`[rdi] := [rsi]  // x86-64`
	Copies the byte, word, or double word pointed at by ESI (x86) or RSI (x86-64) to the memory location pointed at by EDI (x86) or RDI (x86-64). After moving the data, these instructions increment ESI/RSI and EDI/RDI by 1, 2, 4, or 8 if the direction flag is clear; they decrement ESI/RSI and EDI/RDI by 1, 2, 4, or 8 if the direction flag is set.

`rep movsb`
`rep movsw`
`rep movsd`
`rep movsq`
	`[edi] := [esi]  // x86 * count in ECX`
	`[rdi] := [rsi]  // x86-64 * count in RCX`
	Copies the byte, word, or double word pointed at by ESI (x86) or RSI (x86-64) to the memory location pointed at by EDI (x86) or RDI (x86-64). After moving the data, these instructions increment ESI/RSI and EDI/RDI by 1, 2, 4, or 8 if the direction flag is clear; they decrement ESI/RSI and EDI/RDI by 1, 2, 4, or 8 if the direction flag is set. These instructions repeat this operation the number of times specified by the count in the ECX (x86)/RCX (x86-64) register.

A.18 movsx, movzx

The `movsx` and `movzx` instructions require two operands: a source and a destination. The destination operand must be larger than the source operand. These instructions copy the smaller data to the larger object using sign extension (`movsx`) or zero extension (`movzx`). Compilers use these instructions to translate smaller values to larger objects. These instructions do not affect any flags.

NOTE *See* WGC1 *for details on sign extension.*

Table A-58: HLA Syntax for `movsx` and `movzx`

Instruction	Description
`movsx( srcreg, destreg );`	*destreg* := *srcreg*. Sign-extends *srcreg* to the size of *destreg*. *destreg* can be a 16-bit or 32-bit register, *srcreg* must be an 8-bit or 16-bit register and *srcreg* must be smaller than *destreg*.
`movzx( srcreg, destreg );`	*destreg* := *srcreg* Zero-extends *srcreg* to the size of *destreg*. *destreg* can be a 16-bit or 32-bit register; *srcreg* must be an 8-bit or 16-bit register and *srcreg* must be smaller than *destreg*.
`movsx( srcmem, destreg );`	*destreg* := *srcmem* Sign-extends the value of *srcmem* to the size of *destreg*. *destreg* may be any 16-bit or 32-bit general-purpose register; *srcmem* is an 8-bit or 16-bit memory location that is smaller than *destreg*.
`movzx( srcmem, destreg );`	*destreg* := *srcmem* Zero-extends the value of *srcmem* to the size of *destreg*. *destreg* may be any 16-bit or 32-bit general-purpose register; *srcmem* is an 8-bit or 16-bit memory location that is smaller than *destreg*.

Table A-59: Gas Syntax for `movsbw`, `movsbl`, `movslq`, `movswl`, `movzbw`, `movzbl`, `movzlq`, and `movzwl`

Instruction	Description
`movsbw` $srcreg_8$, $destreg_{16}$ `movsbl` $srcreg_8$, $destreg_{32}$ `movswl` $srcreg_{16}$, $destreg_{32}$ `movslq` $srcreg_{32}$, $destreg_{64}$	$destreg_n$:= $srcreg_m$ Sign-extends $srcreg_m$ to the size of $destreg_n$. $destreg_n$ can be a 16-bit, 32-bit, or 64-bit register, as appropriate for the instruction. $srcreg_m$ must be an 8-bit, 16-bit, or 32-bit register, as appropriate for the instruction.
`movzbw` $srcreg_8$, $destreg_{16}$ `movzbl` $srcreg_8$, $destreg_{32}$ `movzwl` $srcreg_{16}$, $destreg_{32}$ `movzlq` $srcreg_{32}$, $destreg_{64}$	$destreg_n$:= $srcreg_m$ Zero-extends $srcreg_m$ to the size of $destreg_n$. $destreg_n$ can be a 16-bit, 32-bit, or 64-bit register, as appropriate for the instruction. $srcreg_m$ must be an 8-bit, 16-bit, or 32-bit register, as appropriate for the instruction.
`movsbw` $srcmem_8$, $destreg_{16}$ `movsbl` $srcmem_8$, $destreg_{32}$ `movswl` $srcmem_{16}$, $destreg_{32}$ `movslq` $srcmem_{32}$, $destreg_{64}$	$destreg_n$:= $srcmem_m$ Sign-extends the value of $srcmem_n$ to the size of $destreg_n$. $destreg_n$ can be a 16-bit, 32-bit, or 64-bit register, as appropriate for the instruction. $srcmem_m$ must be an 8-bit, 16-bit, or 32-bit memory location, as appropriate for the instruction.
`movzbw` $srcmem_8$, $destreg_{16}$ `movzbl` $srcmem_8$, $destreg_{32}$ `movzwl` $srcmem_{16}$, $destreg_{32}$ `movzlq` $srcmem_{32}$, $destreg_{64}$	$destreg_n$:= $srcmem_m$ Zero-extends the value of $srcmem_m$ to the size of $destreg_n$. $destreg_n$ can be a 16-bit, 32-bit, or 64-bit register, as appropriate for the instruction. $srcmem_m$ must be an 8-bit, 16-bit, or 32-bit memory location, as appropriate for the instruction.

Table A-60: MASM Syntax for `movsx` and `movzx`

Instruction	Description
`movsx` *`destreg`*`,` *`srcreg`*	*`destreg`* `:=` *`srcreg`* Sign-extends *`srcreg`* to the size of *`destreg`*. *`destreg`* can be a 16-bit, 32-bit, or 64-bit register; *`srcreg`* must be an 8-bit or 16-bit register; and *`srcreg`* must be smaller than *`destreg`*.
`movzx` *`destreg`*`,` *`srcreg`*	*`destreg`* `:=` *`srcreg`* Zero-extends *`srcreg`* to the size of *`destreg`*. *`destreg`* can be a 16-bit, 32-bit, or 64-bit register; *`srcreg`* must be an 8-bit, 16-bit, or 32-bit register; and *`srcreg`* must be smaller than *`destreg`*.
`movsx` *`destreg`*`,` *`srcmem`*	*`destreg`* `:=` *`srcmem`* Sign-extends the value of *`srcmem`* to the size of *`destreg`*. *`destreg`* may be any 16-bit, 32-bit, or 64-bit general-purpose register; *`srcmem`* is an 8-bit, 16-bit, or 32-bit memory location that is smaller than *`destreg`*.
`movzx` *`destreg`*`,` *`srcmem`*	*`destreg`* `:=` *`srcmem`* Zero-extends the value of *`srcmem`* to the size of *`destreg`*. *`destreg`* may be any 16-bit, 32-bit, or 64-bit general-purpose register; *`srcmem`* is an 8-bit, 16-bit, or 32-bit memory location that is smaller than *`destreg`*.

A.19 mul

The `mul` instruction allows a single operand. If the operand is 8 bits (register or memory), `mul` multiplies the 8-bit value in AL by that operand, producing an unsigned product in AX. If that operand is 16 bits, `mul` multiplies the 16-bit value in AX by the operand, leaving the unsigned product in DX:AX (DX contains the HO word, and AX contains the LO word). If the operand is 32 bits, `mul` multiples the 32-bit quantity in EAX by the operand, leaving the unsigned product in EDX:EAX (EDX contains the HO double word, and EAX contains the LO double word). If the operand is a 64-bit operand, `mul` multiples the 64-bit quantity in RAX by the operand, leaving the unsigned product in RDX:RAX (RDX contains the HO quad word, and RAX contains the LO quad word).

This instruction scrambles the zero and sign flags in the EFLAGS register; you can't rely on flag values after executing a `mul` instruction. It sets the carry and overflow flags if the result doesn't fit into the size specified by the single operand.

Table A-61: HLA Syntax for `mul`

Instruction	Description
`mul(` reg_8 `);`	`ax := al *` reg_8 reg_8 may be any 8-bit general-purpose register.
`mul(` reg_{16} `);`	`dx:ax := ax *` reg_{16} reg_{16} may be any 16-bit general-purpose register.
`mul(` reg_{32} `);`	`edx:eax := eax *` reg_{32} reg_{32} may be any 32-bit general-purpose register.

(continued)

Table A-61: HLA Syntax for `mul` (continued)

Instruction	Description
`mul(` *mem*$_8$ `);`	`ax := al *` *mem*$_8$ *mem*$_8$ may be any 8-bit memory location.
`mul(` *mem*$_{16}$ `);`	`dx:ax := ax *` *mem*$_{16}$ *mem*$_{16}$ may be any 16-bit memory location.
`mul(` *mem*$_{32}$ `);`	`edx:eax := eax *` *mem*$_{32}$ *mem*$_{32}$ may be any 32-bit memory location.

Table A-62: Gas Syntax for `mul`

Instruction	Description
`mulb` *reg*$_8$	`ax := al *` *reg*$_8$ *reg*$_8$ may be any 8-bit general-purpose register.
`mulw` *reg*$_{16}$	`dx:ax := ax *` *reg*$_{16}$ *reg*$_{16}$ may be any 16-bit general-purpose register.
`mull` *reg*$_{32}$	`edx:eax := eax *` *reg*$_{32}$ *reg*$_{32}$ may be any 32-bit general-purpose register.
`mulq` *reg*$_{64}$	`rdx:rax := rax *` *reg*$_{64}$ *reg*$_{64}$ may be any 64-bit general-purpose register.
`mulb` *mem*$_8$	`ax := al *` *mem*$_8$ *mem*$_8$ may be any 8-bit memory location.
`mulw` *mem*$_{16}$	`dx:ax := ax *` *mem*$_{16}$ *mem*$_{16}$ may be any 16-bit memory location.
`mull` *mem*$_{32}$	`edx:eax := eax *` *mem*$_{32}$ *mem*$_{32}$ may be any 32-bit memory location.
`mulq` *mem*$_{64}$	`rdx:rax := rax *` *mem*$_{64}$ *mem*$_{64}$ may be any 64-bit memory location.

Table A-63: MASM Syntax for `mul`

Instruction	Description
`mul` *reg*$_8$	`ax := al *` *reg*$_8$ *reg*$_8$ may be any 8-bit general-purpose register.
`mul` *reg*$_{16}$	`dx:ax := ax *` *reg*$_{16}$ *reg*$_{16}$ may be any 16-bit general-purpose register.
`mul` *reg*$_{32}$	`edx:eax := eax *` *reg*$_{32}$ *reg*$_{32}$ may be any 32-bit general-purpose register.
`mul` *reg*$_{64}$	`rdx:rax := rax *` *reg*$_{64}$ *reg*$_{64}$ may be any 64-bit general-purpose register.
`mul` *mem*$_8$	`ax := al *` *mem*$_8$ *mem*$_8$ may be any 8-bit memory location.
`mul` *mem*$_{16}$	`dx:ax := ax *` *mem*$_{16}$ *mem*$_{16}$ may be any 16-bit memory location.
`mul` *mem*$_{32}$	`edx:eax := eax *` *mem*$_{32}$ *mem*$_{32}$ may be any 32-bit memory location.
`mul` *mem*$_{64}$	`rdx:rax := rax *` *mem*$_{64}$ *mem*$_{64}$ may be any 64-bit memory location.

Table A-64: EFLAGS Settings for `mul`

Flag	Setting
Carry	Set if unsigned overflow occurs.
Overflow	Set if unsigned overflow occurs.
Sign	Scrambled by the `mul` instruction.
Zero	Scrambled by the `mul` instruction.

A.20 neg

The `neg` (negate) instruction requires a single operand. The CPU takes the two's complement of this operand (that is, it negates the value). This instruction also sets several flags in the EFLAGS register, based on the result.

Table A-65: HLA Syntax for `neg`

Instruction	Description
`neg( reg );`	*reg* `:= -` *reg* *reg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
`neg( mem );`	*mem* `:= -` *mem* *mem* may be any 8-bit, 16-bit, or 32-bit memory variable.

Table A-66: Gas Syntax for `neg`

Instruction	Description
`negb` reg_8 `negw` reg_{16} `negl` reg_{32} `negq` reg_{64}	reg_n `:= -` reg_n reg_n may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, as appropriate for the suffix.
`negb` mem_8 `negw` mem_{16} `negl` mem_{32} `negq` mem_{32}	mem_n `:= -` mem_n mem_n may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable, as appropriate for the suffix.

Table A-67: MASM Syntax for `neg`

Instruction	Description
`neg` *reg*	*reg* `:= -` *reg* *reg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register.
`neg` *mem*	*mem* `:= -` *mem* *mem* may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable.

Table A-68: EFLAGS Settings for neg

Flag	Setting
Carry	Set if there is an unsigned overflow.
Overflow	Set if the original value was the smallest negative value (which cannot be negated in the two's complement system).
Sign	Set negation produces a 1 in the HO bit position.
Zero	Set if negation produces 0 (that is, the value was originally 0).

A.21 not

The not instruction requires a single operand. The CPU inverts all the bits in this operand. This instruction also sets several flags in the EFLAGS register, based on the result.

Table A-69: HLA Syntax for not

Instruction	Description
not(*reg*);	*reg* := not *reg* *reg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
not(*mem*);	*mem* := not *mem* *mem* may be any 8-bit, 16-bit, or 32-bit memory variable.

Table A-70: Gas Syntax for not

Instruction	Description
notb reg_8 notw reg_{16} notl reg_{32} notq reg_{64}	reg_n := not reg_n reg_n may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, as appropriate for the suffix.
notb mem_8 notw mem_{16} notl mem_{32} notq mem_{64}	mem_n := not mem_n mem_n may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable, as appropriate for the suffix.

Table A-71: MASM Syntax for not

Instruction	Description
not *reg*	*reg* := not *reg* *reg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register.
not *mem*	*mem* := not *mem* *mem* may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable.

Table A-72: EFLAGS Settings for `not`

Flag	Setting
Carry	Always cleared.
Overflow	Always cleared.
Sign	Set if logical `not` produces a `1` in the HO bit position.
Zero	Set if logical `not` produces `0` (that is, the value was originally all `1` bits).

A.22 or

The `or` instruction requires two operands: a source and a destination. It computes the bitwise logical OR of these two operands' values and stores the result into the destination operand. It also sets several flags in the EFLAGS register, based on the result of the bitwise result.

Table A-73: HLA Syntax for `or`

Instruction	Description
`or(` *`constant`*`,` *`destreg`* `);`	*`destreg`* := *`destreg`* OR *`constant`* *`destreg`* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
`or(` *`constant`*`,` *`destmem`* `);`	*`destmem`* := *`destmem`* OR *`constant`* *`destmem`* may be any 8-bit, 16-bit, or 32-bit memory variable.
`or(` *`srcreg`*`,` *`destreg`* `);`	*`destreg`* := *`destreg`* OR *`srcreg`* *`destreg`* and *`srcreg`* may be any 8-bit, 16-bit, or 32-bit general-purpose registers, though they must both be the same size.
`or(` *`srcmem`*`,` *`destreg`* `);`	*`destreg`* := *`destreg`* OR *`srcmem`* *`destreg`* may be any 8-bit, 16-bit, or 32-bit general-purpose register; *`srcmem`* can be any like-sized memory location.
`or(` *`srcreg`*`,` *`destmem`* `);`	*`destmem`* := *`destmem`* OR *`srcreg`* *`srcreg`* may be any 8-bit, 16-bit, or 32-bit general-purpose register; *`destmem`* can be any like-sized memory location.

Table A-74: Gas Syntax for `or`

Instruction	Description
`orb` *`constant`*`,` *`destreg`*$_8$ `orw` *`constant`*`,` *`destreg`*$_{16}$ `orl` *`constant`*`,` *`destreg`*$_{32}$	*`destreg`*$_n$:= *`destreg`*$_n$ OR *`constant`* *`destreg`*$_n$ may be any 8-bit, 16-bit, or 32-bit general-purpose register, as appropriate for the suffix.
`orb` *`constant`*`,` *`destmem`*$_8$ `orw` *`constant`*`,` *`destmem`*$_{16}$ `orl` *`constant`*`,` *`destmem`*$_{32}$	*`destmem`*$_n$:= *`destmem`*$_n$ OR *`constant`* *`destmem`*$_n$ may be any 8-bit, 16-bit, or 32-bit memory variable, as appropriate for the suffix.
`orb` *`srcreg`*$_8$`,` *`destreg`*$_8$ `orw` *`srcreg`*$_{16}$`,` *`destreg`*$_{16}$ `orl` *`srcreg`*$_{32}$`,` *`destreg`*$_{32}$ `orq` *`srcreg`*$_{64}$`,` *`destreg`*$_{64}$	*`destreg`*$_n$:= *`destreg`*$_n$ OR *`srcreg`*$_n$ *`destreg`*$_n$ and *`srcreg`*$_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose registers, as specified by the suffix.

(continued)

Table A-74: Gas Syntax for `or` (continued)

Instruction	Description
`orb` *srcmem*$_8$, *destreg*$_8$ `orw` *srcmem*$_{16}$, *destreg*$_{16}$ `orl` *srcmem*$_{32}$, *destreg*$_{32}$ `orq` *srcmem*$_{64}$, *destreg*$_{64}$	*destreg*$_n$:= *destreg*$_n$ OR *srcmem*$_n$ *destreg*$_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, according to the suffix; *srcmem*$_n$ can be any like-sized memory location.
`orb` *srcreg*$_8$, *destmem*$_8$ `orw` *srcreg*$_{16}$, *destmem*$_{16}$ `orl` *srcreg*$_{32}$, *destmem*$_{32}$ `orq` *srcreg*$_{64}$, *destmem*$_{64}$	*destmem*$_n$:= *destmem*$_n$ OR *srcreg*$_n$ *srcreg*$_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, as specified by the suffix; *destmem*$_n$ can be any like-sized memory location.

Table A-75: MASM Syntax for `or`

Instruction	Description
`or` *destreg*, *constant*	*destreg* := *destreg* OR *constant* *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
`or` *destmem*, *constant*	*destmem* := *destmem* OR *constant* *destmem* may be any 8-bit, 16-bit, or 32-bit memory variable.
`or` *destreg*, *srcreg*	*destreg* := *destreg* OR *srcreg* *destreg* and *srcreg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose registers, though they must both be the same size.
`or` *destreg*, *srcmem*	*destreg* := *destreg* OR *srcmem* *destreg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, *srcmem* can be any like-sized memory location.
`or` *destmem*, *srcreg*	*destmem* := *destmem* OR *srcreg* *srcreg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register; *destmem* can be any like-sized memory location.

Table A-76: EFLAGS Settings for `or`

Flag	Setting
Carry	Always clear.
Overflow	Always clear.
Sign	Set if the result has a `1` in its HO bit position.
Zero	Set if the result is `0`.

A.23 push, pushfd, pushd, pushq, and pushw

The `push` instruction pushes data onto the 80x86 *hardware stack*, a region in memory that is addressed by the 80x86 ESP (x86) or RSP (x86-64) register. `push` requires a single 16-bit, 32-bit, or 64-bit register, memory, or constant operand and does the following:

1. Subtract the size of the operand in bytes (2, 4, or 8) from the ESP/RSP.

2. Store a copy of the operand's value at the memory location now referenced by ESP/RSP.

The `pushfd` instruction pushes a copy of the 80x86 EFLAGS register onto the stack (4 bytes) or RFLAGS register (8 bytes). Often, you'll see the instructions `pushd` and `pushw` in assembly code. They are used to push double-word or word constants (respectively) onto the stack.

None of these instructions affects any flags in the EFLAGS register.

Table A-77: HLA Syntax for `push`, `pushfd`, `pushw`, and `pushd`

Instruction	Description
`push(` *`constant`* `);` `pushd(` *`constant`* `);`	Pushes a 32-bit value onto the stack.
`pushw(` *`constant`* `);`	Pushes a 16-bit *`constant`* onto the stack.
`push(` *`srcreg`* `);`	Pushes a register onto the stack. *`srcreg`* must be a 16-bit or 32-bit general-purpose register.
`push(` *`srcmem`* `);`	Pushes the contents of a memory location onto the stack. *`srcmem`* must be a 16-bit or 32-bit memory variable.
`pushfd();`	Pushes a copy of the EFLAGS register onto the stack.

Table A-78: Gas Syntax for `pushfl`, `pushfq`, `pushl`, `pushq`, and `pushw`

Instruction	Description
`pushw` *`constant`*	Pushes a 16-bit value onto the stack.
`pushl` *`constant`*	Pushes a 32-bit *`constant`* onto the stack.
`pushw` $srcreg_{16}$ `pushl` $srcreg_{32}$ `pushq` $srcreg_{64}$	Pushes a register onto the stack. $srcreg_n$ must be a 16-bit, 32-bit, or 64-bit general-purpose register, as appropriate for the instruction.
`pushw` $srcmem_{16}$ `pushl` $srcmem_{32}$ `pushq` $srcmem_{64}$	Pushes the contents of a memory location onto the stack. $srcmem_n$ must be a 16-bit, 32-bit, or 64-bit memory variable, as appropriate for the instruction.
`pushfl` `pushfq`	Pushes a copy of the EFLAGS (`pushfl`) or RFLAGS (`pushfq`) register onto the stack.

Table A-79: MASM Syntax for `push`, `pushfd`, `pushfq`, `pushw`, and `pushd`

Instruction	Description
`pushd` *`constant`*	Pushes a 32-bit value onto the stack.
`pushw` *`constant`*	Pushes a 16-bit constant onto the stack.
`push` *`srcreg`*	Pushes a register onto the stack. *`srcreg`* must be a 16-bit, 32-bit, or 64-bit general-purpose register.
`push` *`srcmem`*	Pushes the contents of a memory location onto the stack. *`srcmem`* must be a 16-bit, 32-bit, or 64-bit memory variable.
`pushfd` `pushfq`	Pushes a copy of the EFLAGS (`pushfd`) or RFLAGS (`pushfq`) register onto the stack.

A.24 pop and popfd

The pop instruction removes data pushed onto the 80x86 hardware stack (see the previous section for details on the stack). pop requires a single 16-bit, 32-bit, or 64-bit register or memory operand and does the following:

1. Fetch a copy of the word or double word (depending on pop's operand size) from the memory location pointed at by ESP/RSP and move this data to the location specified by the operand.
2. Add the size of the operand in bytes (2, 4, or 8) to the ESP/RSP register.

The popfd instruction pops the double word on the stack into the 80x86 EFLAGS register.

The pop instruction does not affect any flags in the EFLAGS register; however, popfd replaces all the flags with the value read from the stack.

Table A-80: HLA Syntax for pop and popfd

Instruction	Description
pop(*destreg*);	Pops a value from the stack into a register. *destreg* must be a 16-bit or 32-bit general-purpose register.
pop(*destmem*);	Pops a value from the stack into a memory variable. *destmem* must be a 16-bit or 32-bit memory variable.
popfd();	Pops the double word on the stack into the EFLAGS register.

Table A-81: Gas Syntax for pop and popfd

Instruction	Description
popw $destreg_{16}$ popl $destreg_{32}$ popq $destreg_{64}$	Pops a value from the stack into a register. $destreg_n$ must be a 16-bit, 32-bit, or 64-bit general-purpose register, as appropriate for the instruction.
popw $destmem_{16}$ popl $destmem_{32}$ popq $destmem_{64}$	Pops a value from the stack into a memory variable. $destmem_n$ must be a 16-bit, 32-bit, or 64-bit memory variable, as appropriate for the instruction.
popfl popfq	Pops the EFLAGS (popfl) or RFLAGS (popfq) register from the stack.

Table A-82: MASM Syntax for pop and popfd

Instruction	Description
pop *destreg*	Pops a value from the stack into a register. *destreg* must be a 16-bit or 32-bit general-purpose register.
pop *destmem*	Pops a value from the stack into a memory variable. *destmem* must be a 16-bit or 32-bit memory variable.
popfd popfq	Pops the double word on the stack into the EFLAGS register (popfd) or the quad word on the stack into the RFLAGS register (popfq).

Table A-83: EFLAGS Settings for `popfd`

Flag	Setting
Carry	Set according to the value popped from the stack.
Overflow	Set according to the value popped from the stack.
Sign	Set according to the value popped from the stack.
Zero	Set according to the value popped from the stack.

A.25 ret

The `ret` instruction returns control from a subroutine. There are two forms of this instruction—one with no operand and one with a single constant operand. Both forms pop an address from the stack and transfer control to the location specified by this *return address*. This is generally an address pushed onto the stack by a `call` instruction. The `ret` instruction with a 16-bit constant operand also zero-extends the value to 32/64 bits and adds it to the ESP/RSP register (after popping the return address from the stack). This form automatically removes parameters passed on the stack by the calling code. These two instructions do not affect any flags in the EFLAGS register.

Table A-84: HLA Syntax for `ret`

Instruction	Description
`ret();`	Pops a return address from the stack and transfers control to that return address.
`ret(` *constant*$_{16}$ `);`	Pops a return address from the stack, adds the *constant* operand's value to the ESP register, and then transfers control to the return address.

Table A-85: Gas Syntax for `ret`

Instruction	Description
`ret`	Pops a return address from the stack and transfers control to that return address.
`ret` *constant*$_{16}$	Pops a return address from the stack, adds the *constant* operand's value to the ESP/RSP register, and then transfers control to the return address.

Table A-86: MASM Syntax for `ret`

Instruction	Description
`ret`	Pops a return address from the stack and transfers control to that return address.
`ret` *constant*$_{16}$	Pops a return address from the stack, adds the *constant* operand's value to the ESP/RSP register, and then transfers control to the return address.

A.26 sar, shr, shl

The sar, shr, and shl instructions require two operands: a count and a destination. These instructions *shift* the destination operand count bits to the left or right (depending on the instruction).

The sar (*shift arithmetic right*) instruction copies all the bits in the destination operand from HO bit positions to LO bit positions, the number of positions specified by the count operand. The last bit shifted out of the LO bit position is shifted into the carry flag. The HO bit is unaffected by the sar instruction.

The shr (*shift right*, or *shift logical right*) instruction copies all the bits in the destination operand from HO bit positions to LO bit positions by the number of positions specified by the count operand. The last bit shifted out of the LO bit position is shifted into the carry flag. This instruction shifts a 0 into the HO bit position after each bit shift occurs.

The shl (*shift left*, or s*hift logical left*) instruction copies all the bits in the destination operand from LO bit positions to HO bit positions by the number of positions specified by the count operand. The last bit shifted out of the HO bit position is shifted into the carry flag. This instruction shifts a 0 into the LO bit position after each shift operation.

The count operand can either be an immediate constant or the CL register.

Table A-87: HLA Syntax for shl, shr, and sar

Instruction	Description
shl(*constant*, *destreg*);	*destreg* := *destreg* SHL *constant* *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
shl(*constant*, *destmem*);	*destmem* := *destmem* SHL *constant* *destmem* may be any 8-bit, 16-bit, or 32-bit memory variable.
shl(*cl*, *destreg*);	*destreg* := *destreg* SHL *cl* *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
shl(*cl*, *destmem*);	*destmem* := *destmem* SHL *cl* *destmem* may be any 8-bit, 16-bit, or 32-bit memory variable.
shr(*constant*, *destreg*);	*destreg* := *destreg* SHR *constant* *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
shr(*constant*, *destmem*);	*destmem* := *destmem* SHR *constant* *destmem* may be any 8-bit, 16-bit, or 32-bit memory variable.
shr(*cl*, *destreg*);	*destreg* := *destreg* SHR *cl* *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
shr(*cl*, *destmem*);	*destmem* := *destmem* SHR *cl* *destmem* may be any 8-bit, 16-bit, or 32-bit memory variable.

(continued)

Table A-87: HLA Syntax for `shl`, `shr`, and `sar` (continued)

Instruction	Description
`sar( constant, destreg );`	*destreg* := *destreg* SAR *constant* *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
`sar( constant, destmem );`	*destmem* := *destmem* SAR *constant* *destmem* may be any 8-bit, 16-bit, or 32-bit memory variable.
`sar( cl, destreg );`	*destreg* := *destreg* SAR *cl* *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
`sar( cl, destmem );`	*destmem* := *destmem* SAR *cl* *destmem* may be any 8-bit, 16-bit, or 32-bit memory variable.

Table A-88: Gas Syntax for `shl`, `sar`, and `shr`

Instruction	Description
`shlb` *constant*, *destreg*$_8$ `shlw` *constant*, *destreg*$_{16}$ `shll` *constant*, *destreg*$_{32}$ `shlq` *constant*, *destreg*$_{64}$	*destreg*$_n$:= *destreg*$_n$ SHL *constant* *destreg*$_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, as appropriate for the instruction.
`shlb` *constant*, *destmem*$_8$ `shlw` *constant*, *destmem*$_{16}$ `shll` *constant*, *destmem*$_{32}$ `shlq` *constant*, *destmem*$_{64}$	*destmem*$_n$:= *destmem*$_n$ SHL *constant* *destmem*$_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable, as appropriate for the instruction.
`shlb %cl,` *destreg*$_8$ `shlw %cl,` *destreg*$_{16}$ `shll %cl,` *destreg*$_{32}$ `shlq %cl,` *destreg*$_{64}$	*destreg*$_n$:= *destreg*$_n$ SHL *cl* *destreg*$_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, as appropriate for the instruction.
`shlb %cl,` *destmem*$_8$ `shlw %cl,` *destmem*$_{16}$ `shll %cl,` *destmem*$_{32}$ `shlq %cl,` *destmem*$_{64}$	*destmem*$_n$:= *destmem*$_n$ SHL *cl* *destmem*$_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable, as appropriate for the instruction.
`shrb` *constant*, *destreg*$_8$ `shrw` *constant*, *destreg*$_{16}$ `shrl` *constant*, *destreg*$_{32}$ `shrq` *constant*, *destreg*$_{64}$	*destreg*$_n$:= *destreg*$_n$ SHR *constant* *destreg*$_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, as appropriate for the instruction.
`shrb` *constant*, *destmem*$_8$ `shrw` *constant*, *destmem*$_{16}$ `shrl` *constant*, *destmem*$_{32}$ `shrq` *constant*, *destmem*$_{64}$	*destmem*$_n$:= *destmem*$_n$ SHR *constant* *destmem*$_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable, as appropriate for the instruction.
`shrb %cl,` *destreg*$_8$ `shrw %cl,` *destreg*$_{16}$ `shrl %cl,` *destreg*$_{32}$ `shrq %cl,` *destreg*$_{64}$	*destreg*$_n$:= *destreg*$_n$ SHR *cl* *destreg*$_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, as appropriate for the instruction.
`shrb %cl,` *destmem*$_8$ `shrw %cl,` *destmem*$_{16}$ `shrl %cl,` *destmem*$_{32}$ `shrq %cl,` *destmem*$_{64}$	*destmem*$_n$:= *destmem*$_n$ SHR *cl* *destmem*$_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable, as appropriate for the instruction.

(continued)

Table A-88: Gas Syntax for `shl`, `sar`, and `shr` (continued)

Instruction	Description
`sarb` *constant*`,` *destreg*$_8$ `sarw` *constant*`,` *destreg*$_{16}$ `sarl` *constant*`,` *destreg*$_{32}$ `sarq` *constant*`,` *destreg*$_{64}$	*destreg*$_n$:= *destreg*$_n$ SAR *constant* *destreg*$_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, as appropriate for the instruction.
`sarb` *constant*`,` *destmem*$_8$ `sarw` *constant*`,` *destmem*$_{16}$ `sarl` *constant*`,` *destmem*$_{32}$ `sarq` *constant*`,` *destmem*$_{64}$	*destmem*$_n$:= *destmem*$_n$ SAR *constant* *destmem*$_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable, as appropriate for the instruction.
`sarb %cl,` *destreg*$_8$ `sarw %cl,` *destreg*$_{16}$ `sarl %cl,` *destreg*$_{32}$ `sarq %cl,` *destreg*$_{64}$	*destreg*$_n$:= *destreg*$_n$ SAR *cl* *destreg*$_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, as appropriate for the instruction.
`sarb %cl,` *destmem*$_8$ `sarlw %cl,` *destmem*$_{16}$ `sarl %cl,` *destmem*$_{32}$ `sarq %cl,` *destmem*$_{64}$	*destmem*$_n$:= *destmem*$_n$ SAR *cl* *destmem*$_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable, as appropriate for the instruction.

Table A-89: MASM Syntax for `shl`, `sar`, and `shr`

Instruction	Description
`shl` *destreg*`,` *constant*	*destreg* := *destreg* SHL *constant* *destreg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register.
`shl` *destmem*`,` *constant*	*destmem* := *destmem* SHL *constant* *destmem* may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable.
`shl` *destreg*`,` *cl*	*destreg* := *destreg* SHL *cl* *destreg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register.
`shl` *destmem*`,` *cl*	*destmem* := *destmem* SHL *cl* *destmem* may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable.
`shr` *destreg*`,` *constant*	*destreg* := *destreg* SHR *constant* *destreg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register.
`shr` *destmem*`,` *constant*	*destmem* := *destmem* SHR *constant* *destmem* may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable.
`shr` *destreg*`,` *cl*	*destreg* := *destreg* SHR *cl* *destreg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register.
`shr` *destmem*`,` *cl*	*destmem* := *destmem* SHR *cl* *destmem* may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable.
`sar` *destreg*`,` *constant*	*destreg* := *destreg* SAR *constant* *destreg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register.

(continued)

Table A-89: MASM Syntax for `shl`, `sar`, and `shr` (continued)

Instruction	Description
`sar` *`destmem`*`,` *`constant`*	*`destmem`* `:=` *`destmem`* `SAR` *`constant`* *`destmem`* may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable.
`sar` *`destreg`*`,` *`cl`*	*`destreg`* `:=` *`destreg`* `SAR` *`cl`* *`destreg`* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register.
`sar` *`destmem`*`,` *`cl`*	*`destmem`* `:=` *`destmem`* `SAR` *`cl`* *`destmem`* may be any 8-bit, 16-bit, 32-bit, or 64-bit memory variable.

Table A-90: EFLAGS Settings for `shl`, `sar`, and `shr`

Flag	Setting
Carry	Contains the last bit shifted out of the LO bit position (`shr`, `sar`) or the HO bit position (`shl`).
Overflow	Set if the HO two bits change their values during the shift.
Sign	Set if the result has a `1` in its HO bit position.
Zero	Set if the result is `0`.

A.27 Conditional Set Instructions

The 80x86 supports a wide variety of conditional set (`scc`) instructions that set an 8-bit register or memory location to `0` or `1` based upon tests on the EFLAGS register. These instructions allow the CPU to set the Boolean variables `true` or `false` based on conditions computed by instructions, such as `cmp`, that affect the EFLAGS register. Note, however, that these instructions do not themselves modify the EFLAGS register.

Table A-91: Conditional Set Instructions

Instruction	Description

HLA:
`seta(` reg_8 `);`
`seta(` mem_8 `);`
 	
MASM/Gas:
`seta` reg_8
`seta` mem_8
	Unsigned conditional set if above (carry = `0` and zero = `0`). Stores a `1` in the destination operand if the result of the previous comparison found the first operand to be greater than the second using an unsigned comparison. Stores a `0` into the destination operand otherwise.

HLA:
`setae(` reg_8 `);`
`setae(` mem_8 `);`
 	
MASM/Gas:
`setae` reg_8
`setae` mem_8
	Unsigned conditional set if above or equal (carry = `0`). See `seta` for details.

(continued)

Table A-91: Conditional Set Instructions (continued)

Instruction	Description

HLA:
`setb( reg8 );`
`setb( mem8 );`
 	
MASM/Gas:
`setb reg8`
`setb mem8`
	Unsigned conditional set if below (carry = 1). See `seta` for details.

HLA:
`setbe( reg8 );`
`setbe( mem8 );`
 	
MASM/Gas:
`setbe reg8`
`setbe mem8`
	Unsigned conditional set if below or equal (carry = 1 or zero = 1). See `seta` for details.

HLA:
`setc( reg8 );`
`setc( mem8 );`
 	
MASM/Gas:
`setc reg8`
`setc mem8`
	Conditional set if carry set (carry = 1). See `seta` for details.

HLA:
`sete( reg8 );`
`sete( mem8 );`
 	
MASM/Gas:
`sete reg8`
`sete mem8`
	Conditional set if equal (zero = 1). See `seta` for details.

HLA:
`setg( reg8 );`
`setg( mem8 );`
 	
MASM/Gas:
`setg reg8`
`setg mem8`
	Signed conditional set if greater (sign = overflow and zero=0). See `seta` for details.

HLA:
`setge( reg8 );`
`setge( mem8 );`
 	
MASM/Gas:
`setge reg8`
`setge mem8`
	Signed conditional set if greater or equal (sign = overflow or zero = 1). See `seta` for details.

HLA:
`setl( reg8 );`
`setl( mem8 );`
 	
MASM/Gas:
`setl reg8`
`setl mem8`
	Signed conditional set if less than (sign <> overflow). See `seta` for details.

(continued)

Table A-91: Conditional Set Instructions (continued)

Instruction	Description

HLA:
`setle( reg8 );`
`setle( mem8 );`
 	
MASM/Gas:
`setle reg8`
`setle mem8`
	Signed conditional set if less than or equal (sign <> overflow or zero = 1). See `seta` for details.

HLA:
`setna( reg8 );`
`setna( mem8 );`
 	
MASM/Gas:
`setna reg8`
`setna mem8`
	Unsigned conditional set if not above (carry = 1 or zero = 1). See `seta` for details.

HLA:
`setnae( reg8 );`
`setnae( mem8 );`
 	
MASM/Gas:
`setnae reg8`
`setnae mem8`
	Unsigned conditional set if not above or equal (carry = 1). See `seta` for details.

HLA:
`setnb( reg8 );`
`setnb( mem8 );`
 	
MASM/Gas:
`setnb reg8`
`setnb mem8`
	Unsigned conditional set if not below (carry = 0). See `seta` for details.

HLA:
`setnbe( reg8 );`
`setnbe( mem8 );`
 	
MASM/Gas:
`setnbe reg8`
`setnbe mem8`
	Unsigned conditional set if not below or equal (carry = 0 and zero = 0). See `seta` for details.

HLA:
`setnc( reg8 );`
`setnc( mem8 );`
 	
MASM/Gas:
`setnc reg8`
`setnc mem8`
	Conditional set if carry clear (carry = 0). See `seta` for details.

HLA:
`setne( reg8 );`
`setne( mem8 );`
 	
MASM/Gas:
`setne reg8`
`setne mem8`
	Conditional set if not equal (zero = 0). See `seta` for details.

(continued)

Table A-91: Conditional Set Instructions (continued)

Instruction	Description

HLA:
`setng(` *reg*$_8$ `);`
`setng(` *mem*$_8$ `);`
 	
MASM/Gas:
`setng` *reg*$_8$
`setng` *mem*$_8$
	Signed conditional set if not greater (sign <> overflow or zero = 1). See `seta` for details.

HLA:
`setnge(` *reg*$_8$ `);`
`setnge(` *mem*$_8$ `);`
 	
MASM/Gas:
`setnge` *reg*$_8$
`setnge` *mem*$_8$
	Signed conditional set if not greater than (sign <> overflow). See `seta` for details.

HLA:
`setnl(` *reg*$_8$ `);`
`setnl(` *mem*$_8$ `);`
 	
MASM/Gas:
`setnl` *reg*$_8$
`setnl` *mem*$_8$
	Signed conditional set if not less than (sign = overflow or zero = 1). See `seta` for details.

HLA:
`setnle(` *reg*$_8$ `);`
`setnle(` *mem*$_8$ `);`
 	
MASM/Gas:
`setnle` *reg*$_8$
`setnle` *mem*$_8$
	Signed conditional set if not less than or equal (sign = overflow and zero = 0). See `seta` for details.

HLA:
`setno(` *reg*$_8$ `);`
`setno(` *mem*$_8$ `);`
 	
MASM/Gas:
`setno` *reg*$_8$
`setno` *mem*$_8$
	Conditional set if no overflow (overflow = 0). See `seta` for details.

HLA:
`setns(` *reg*$_8$ `);`
`setns(` *mem*$_8$ `);`
 	
MASM/Gas:
`setns` *reg*$_8$
`setns` *mem*$_8$
	Conditional set if no sign (sign = 0). See `seta` for details.

HLA:
`setnz(` *reg*$_8$ `);`
`setnz(` *mem*$_8$ `);`
 	
MASM/Gas:
`setnz` *reg*$_8$
`setnz` *mem*$_8$
	Conditional set if not 0 (zero = 0). See `seta` for details.

(continued)

Table A-91: Conditional Set Instructions (continued)

Instruction	Description

HLA:
`seto(` *reg*$_8$ `);`
`seto(` *mem*$_8$ `);`
 	
MASM/Gas:
`seto` *reg*$_8$
`seto` *mem*$_8$
	Conditional set if overflow (overflow = 1). See `seta` for details.

HLA:
`sets(` *reg*$_8$ `);`
`sets(` *mem*$_8$`);`
 	
MASM/Gas:
`sets` *reg*$_8$
`sets` *mem*$_8$
	Conditional set if sign set (sign = 1). See `seta` for details.

HLA:
`setz(` *reg*$_8$ `);`
`setz(` *mem*$_8$ `);`
 	
MASM/Gas:
`setz` *reg*$_8$
`setz` *mem*$_8$
	Conditional set if `0` (zero = 1). See `seta` for details.

A.28 stos, stosb, stosd, stosq, stosw

The `stos` instructions—the *store string* instructions that copy a value in AL, AX, EAX, or RAX into a range of memory locations—do not require any explicit operands. These instructions take two forms: the store string instruction by itself, or a store string instruction with a "repeat" prefix.

Without a repeat prefix, these instructions copy the value in AL (`stosb`), AX (`stosw`), EAX (`stosd`), or RAX (`stosq`) to the memory location pointed at by EDI (x86) or RDI (x86-64), the destination index register. After copying the data, the CPU either increments or decrements EDI/RDI by the size, in bytes, of the transfer. That is, `stosb` increments or decrements EDI/RDI by 1, `stosw` increments or decrements EDI/RDI by 2, `stosd` increments or decrements EDI/RDI by 4, and `stosq` increments or decrements RDI by 8. These instructions determine whether to increment or decrement EDI/RDI based on the value of the *direction flag* in the EFLAGS register. If the direction flag is clear, the store string instructions increments EDI/RDI; if the direction flag is set, the store string instruction decrements EDI/RDI.

If the repeat prefix is attached to one of these store string instructions, then the CPU repeats the store operation the number of times specified by the ECX (x86) or RCX (x86-64) register. Compilers typically use this instruction to clear out a block of bytes in memory (that is, set the block of bytes to all `0`s).

These instructions do not affect any flags.

Table A-92: HLA Syntax for `stosb`, `stosd`, and `stosw`

Instruction	Description

`stosb();`
`stosw();`
`stosd();`
	`[edi] := AL`
	`[edi] := AX`
	`[edi] := EAX`
	Copies the byte, word, or double word held in AL/AX/EAX to the memory location pointed at by EDI. After moving the data, these instructions increment EDI by 1, 2, or 4 if the direction flag is clear; they decrement EDI by 1, 2, or 4 if the direction flag is set.

`rep.stosb();`
`rep.stosw();`
`rep.stosd();`
	`[edi]..[edi+ecx-1] := AL/AX/EAX`
	Copies the value in AL, AX, or EAX to a block of ECX bytes, words, or double words in memory, where EDI points. Increments or decrements EDI after each movement by the size of the data moved, based on the value of the direction flag.

Table A-93: Gas Syntax for `stosb`, `stosl`, and `stosw`

Instruction	Description

`stosb`
`stosw`
`stosl`
`stosq`
	`[edi] := AL // RDI on x86-64`
	`[edi] := AX // RDI on x86-64`
	`[edi] := EAX // RDI on x86-64`
	`[rdi] := RAX // RDI only on x86-64`
	Copies the byte, word, double word, or quad word held in AL/AX/EAX/RAX to the memory location pointed at by EDI/RDI. After moving the data, these instructions increment EDI/RDI by 1, 2, 4, or 8 if the direction flag is clear; they decrement EDI/RDI by 1, 2, 4, or 8 if the direction flag is set.

`rep movsb`
`rep movsw`
`rep movsd`
`rep movsq`
	`[edi]..[edi+ecx-1] := AL/AX/EAX on x86`
	`[rdi]..[rdi+rcx-1] := AL/AX/EAX/RAX on x86-64`
	Copies the value in AL, AX, EAX, or RAX to a block of ECX (x86) or RCX (x86-64) bytes, words, double words, or quad words in memory, where EDI/RDI points. Increments or decrements EDI/RDI after each movement by the size of the data moved, based on the value of the direction flag.

Table A-94: MASM Syntax for `stosb`, `stosd`, `stosq`, and `stosw`

Instruction	Description

`stosb`
`stosw`
`stosd`
`stosq`
	`[edi] := AL // RDI on x86-64`
	`[edi] := AX // RDI on x86-64`
	`[edi] := EAX // RDI on x64-64`
	`[rdi] := RAX // RDI only on x86-64`
	Copies the byte, word, or double word held in AL/AX/EAX to the memory location pointed at by EDI. After moving the data, these instructions increment EDI by 1, 2, or 4 if the direction flag is clear; they decrement EDI by 1, 2, or 4 if the direction flag is set.

(continued)

Table A-94: MASM Syntax for `stosb`, `stosd`, and `stosw` (continued)

Instruction	Description

`rep stosb`
`rep stosw`
`rep stosd`
`rep stosq`
	`[edi]..[edi+ecx-1] := AL/AX/EAX`
	`[rdi]..[rdi+ecx-1] := AL/AX/EAX/RAX (x86-64 only)`
	Copies the value in AL, AX, or EAX to a block of ECX bytes (RCX on x86-64), words, double words, or quad words in memory, where EDI/RDI points. Increments or decrements EDI/RDI after each movement by the size of the data moved, based on the value of the direction flag.

A.29 sub

The `sub` instruction requires two operands: a source and a destination. It computes the difference of the values of these two operands and stores the difference back into the destination operand. It also sets several flags in the EFLAGS register, based on the result of the subtraction operation (note that `sub` affects the flags exactly the same way as the `cmp` instruction).

Table A-95: HLA Syntax for `sub`

Instruction	Description
`sub( constant, destreg );`	*destreg* := *destreg* – *constant* *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
`sub( constant, destmem );`	*destmem* := *destmem* – *constant* *destmem* may be any 8-bit, 16-bit, or 32-bit memory variable.
`sub( srcreg, destreg );`	*destreg* := *destreg* – *srcreg* *destreg* and *srcreg* may be any 8-bit, 16-bit, or 32-bit general-purpose registers, though they must both be the same size.
`sub( srcmem, destreg );`	*destreg* := *destreg* – *srcmem* *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register; *srcmem* can be any like-sized memory location.
`sub( srcreg, destmem );`	*destmem* := *destmem* – *srcreg* *srcreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register; *destmem* can be any like-sized memory location.

Table A-96: Gas Syntax for `sub`

Instruction	Description
`subb` *constant*, *destreg*$_8$ `subw` *constant*, *destreg*$_{16}$ `subl` *constant*, *destreg*$_{32}$	*destreg*$_n$:= *destreg*$_n$ – *constant* *destreg*$_n$ may be any 8-bit, 16-bit, or 32-bit general-purpose register, as appropriate for the suffix.
`subb` *constant*, *destmem*$_8$ `subw` *constant*, *destmem*$_{16}$ `subl` *constant*, *destmem*$_{32}$	*destmem*$_n$:= *destmem*$_n$ – *constant* *destmem*$_n$ may be any 8-bit, 16-bit, or 32-bit memory variable, as appropriate for the suffix.

(continued)

Table A-96: Gas Syntax for sub (continued)

Instruction	Description

subb $srcreg_8$, $destreg_8$
subw $srcreg_{16}$, $destreg_{16}$
subl $srcreg_{32}$, $destreg_{32}$
subq $srcreg_{64}$, $destreg_{64}$
	$destreg_n$:= $destreg_n$ – $srcreg_n$
	$destreg_n$ and $srcreg_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose registers, as specified by the suffix.

subb $srcmem_8$, $destreg_8$
subw $srcmem_{16}$, $destreg_{16}$
subl $srcmem_{32}$, $destreg_{32}$
subq $srcmem_{64}$, $destreg_{64}$
	$destreg_n$:= $destreg_n$ – $srcmem_n$
	$destreg_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, according to the suffix; $srcmem_n$ can be any like-sized memory location.

subb $srcreg_8$, $destmem_8$
subw $srcreg_{16}$, $destmem_{16}$
subl $srcreg_{32}$, $destmem_{32}$
subq $srcreg_{64}$, $destmem_{64}$
	$destmem_n$:= $destmem_n$ – $srcreg_n$
	$srcreg_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, as specified by the suffix; $destmem_n$ can be any like-sized memory location.

Table A-97: MASM Syntax for sub

Instruction	Description
sub *destreg*, *constant*	*destreg* := *destreg* – *constant* *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
sub *destmem*, *constant*	*destmem* := *destmem* – *constant* *destmem* may be any 8-bit, 16-bit, or 32-bit memory variable.
sub *destreg*, *srcreg*	*destreg* := *destreg* – *srcreg* *destreg* and *srcreg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose registers, though they must both be the same size.
sub *destreg*, *srcmem*	*destreg* := *destreg* – *srcmem* *destreg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, *srcmem* can be any like-sized memory location.
sub *destmem*, *srcreg*	*destmem* := *destmem* – *srcreg* *srcreg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, *destmem* can be any like-sized memory location.

Table A-98: EFLAGS Settings for sub

Flag	Setting
Carry	Set if the difference of the two values produces an unsigned overflow.
Overflow	Set if the difference of the two values produces a signed overflow.
Sign	Set if the difference of the two values has a 1 in its HO bit position.
Zero	Set if the difference of the two values is 0.

A.30 test

The test instruction requires two operands: a source and a destination. It computes the logical AND of the values of these two operands but only updates the EFLAGS register; it does not store the result of the logical AND operation into either of the two operands. Note that test sets the flags exactly the same way as the and instruction and is often used as an efficient way to test a register to see if it contains 0 (by ANDing that register with itself). It is also often used to test to see if a particular bit in a binary value is set or clear.

Table A-99: HLA Syntax for test

Instruction	Description
test(*constant*, *destreg*);	*destreg* AND *constant* (result to EFLAGS) *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
test(*constant*, *destmem*);	*destmem* – *constant* (result to EFLAGS) *destmem* may be any 8-bit, 16-bit, or 32-bit memory variable.
test(*srcreg*, *destreg*);	*destreg* – *srcreg* (result to EFLAGS) *destreg* and *srcreg* may be any 8-bit, 16-bit, or 32-bit general-purpose registers, though they must both be the same size.
test(*srcmem*, *destreg*);	*destreg* – *srcmem* (result to EFLAGS) *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register; *srcmem* can be any like-sized memory location.
test(*srcreg*, *destmem*);	*destmem* – *srcreg* (result to EFLAGS) *srcreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register; *destmem* can be any like-sized memory location.

Table A-100: Gas Syntax for test

Instruction	Description
testb *constant*, $destreg_8$ testw *constant*, $destreg_{16}$ testl *constant*, $destreg_{32}$	$destreg_n$ – *constant* (result to EFLAGS) $destreg_n$ may be any 8-bit, 16-bit, or 32-bit general-purpose register, as appropriate for the suffix.
testb *constant*, $destmem_8$ testw *constant*, $destmem_{16}$ testl *constant*, $destmem_{32}$	$destmem_n$ – *constant* (result to EFLAGS) $destmem_n$ may be any 8-bit, 16-bit, or 32-bit memory variable, as appropriate for the suffix.
testb $srcreg_8$, $destreg_8$ testw $srcreg_{16}$, $destreg_{16}$ testl $srcreg_{32}$, $destreg_{32}$ testq $srcreg_{64}$, $destreg_{64}$	$destreg_n$ – $srcreg_n$ (result to EFLAGS) $destreg_n$ and $srcreg_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose registers, as specified by the suffix.
testb $srcmem_8$, $destreg_8$ testw $srcmem_{16}$, $destreg_{16}$ testl $srcmem_{32}$, $destreg_{32}$ testq $srcmem_{64}$, $destreg_{64}$	$destreg_n$ – $srcmem_n$ (result to EFLAGS) $destreg_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, according to the suffix; $srcmem_n$ can be any like-sized memory location.
testb $srcreg_8$, $destmem_8$ testw $srcreg_{16}$, $destmem_{16}$ testl $srcreg_{32}$, $destmem_{32}$ testq $srcreg_{64}$, $destmem_{64}$	$destmem_n$ – $srcreg_n$ (result to EFLAGS) $srcreg_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, as specified by the suffix; $destmem_n$ can be any like-sized memory location.

Table A-101: MASM Syntax for `test`

Instruction	Description
`test` *`destreg`*`,` *`constant`*	*`destreg`* – *`constant`* (result to EFLAGS) *`destreg`* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
`test` *`destmem`*`,` *`constant`*	*`destmem`* – *`constant`* (result to EFLAGS) *`destmem`* may be any 8-bit, 16-bit, or 32-bit memory variable.
`test` *`destreg`*`,` *`srcreg`*	*`destreg`* – *`srcreg`* (result to EFLAGS) *`destreg`* and *`srcreg`* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose registers, though they must both be the same size.
`test` *`destreg`*`,` *`srcmem`*	*`destreg`* – *`srcmem`* (result to EFLAGS) *`destreg`* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register; *`srcmem`* can be any like-sized memory location.
`test` *`destmem`*`,` *`srcreg`*	*`destmem`* – *`srcreg`* (result to EFLAGS) *`srcreg`* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register; *`destmem`* can be any like-sized memory location.

Table A-102: EFLAGS Settings for `test`

Flag	Setting
Carry	Cleared
Overflow	Cleared
Sign	Set if the logical AND of the two operands has a `1` in the HO bit position.
Zero	Set if the logical AND of the two operands produces a `0` result.

A.31 xor

The `xor` instruction requires two operands: a source and a destination. It computes the exclusive-OR of the values of these two operands and stores the result back into the destination operand. It also sets several flags in the EFLAGS register, based on the result of the exclusive-OR operation.

Table A-103: HLA Syntax for `xor`

Instruction	Description
`xor(` *`constant`*`,` *`destreg`* `);`	*`destreg`* := *`destreg`* XOR *`constant`* *`destreg`* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
`xor(` *`constant`*`,` *`destmem`* `);`	*`destmem`* := *`destmem`* XOR *`constant`* *`destmem`* may be any 8-bit, 16-bit, or 32-bit memory variable.

(continued)

Table A-103: HLA Syntax for `xor` (continued)

Instruction	Description
`xor( srcreg, destreg );`	*destreg* := *destreg* XOR *srcreg* *destreg* and *srcreg* may be any 8-bit, 16-bit, or 32-bit general-purpose registers, though they must both be the same size.
`xor( srcmem, destreg );`	*destreg* := *destreg* XOR *srcmem* *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register; *srcmem* can be any like-sized memory location.
`xor( srcreg, destmem );`	*destmem* := *destmem* XOR *srcreg* *srcreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register; *destmem* can be any like-sized memory location.

Table A-104: Gas Syntax for `xor`

Instruction	Description
`xorb` *constant*, *destreg*$_8$ `xorw` *constant*, *destreg*$_{16}$ `xorl` *constant*, *destreg*$_{32}$	*destreg*$_n$:= *destreg*$_n$ XOR *constant* *destreg*$_n$ may be any 8-bit, 16-bit, or 32-bit general-purpose register, as appropriate for the suffix.
`xorb` *constant*, *destmem*$_8$ `xorw` *constant*, *destmem*$_{16}$ `xorl` *constant*, *destmem*$_{32}$	*destmem*$_n$:= *destmem*$_n$ XOR *constant* *destmem*$_n$ may be any 8-bit, 16-bit, or 32-bit memory variable, as appropriate for the suffix.
`xorb` *srcreg*$_8$, *destreg*$_8$ `xorw` *srcreg*$_{16}$, *destreg*$_{16}$ `xorl` *srcreg*$_{32}$, *destreg*$_{32}$ `xorq` *srcreg*$_{64}$, *destreg*$_{64}$	*destreg*$_n$:= *destreg*$_n$ XOR *srcreg*$_n$ *destreg*$_n$ and *srcreg*$_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose registers, as specified by the suffix.
`xorb` *srcmem*$_8$, *destreg*$_8$ `xorw` *srcmem*$_{16}$, *destreg*$_{16}$ `xorl` *srcmem*$_{32}$, *destreg*$_{32}$ `xorq` *srcmem*$_{64}$, *destreg*$_{64}$	*destreg*$_n$:= *destreg*$_n$ XOR *srcmem*$_n$ *destreg*$_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, according to the suffix; *srcmem*$_n$ can be any like-sized memory location.
`xorb` *srcreg*$_8$, *destmem*$_8$ `xorw` *srcreg*$_{16}$, *destmem*$_{16}$ `xorl` *srcreg*$_{32}$, *destmem*$_{32}$ `xorq` *srcreg*$_{64}$, *destmem*$_{64}$	*destmem*$_n$:= *destmem*$_n$ XOR *srcreg*$_n$ *srcreg*$_n$ may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register, as specified by the suffix; *destmem*$_n$ can be any like-sized memory location.

Table A-105: MASM Syntax for `xor`

Instruction	Description
`xor` *destreg*, *constant*	*destreg* := *destreg* XOR *constant* *destreg* may be any 8-bit, 16-bit, or 32-bit general-purpose register.
`xor` *destmem*, *constant*	*destmem* := *destmem* XOR *constant* *destmem* may be any 8-bit, 16-bit, or 32-bit memory variable.
`xor` *destreg*, *srcreg*	*destreg* := *destreg* XOR *srcreg* *destreg* and *srcreg* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose registers, though they must both be the same size.

(continued)

Table A-105: MASM Syntax for `xor` (continued)

Instruction	Description
`xor` *`destreg`*`,` *`srcmem`*	*`destreg`* `:=` *`destreg`* XOR *`srcmem`* *`destreg`* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register; *`srcmem`* can be any like-sized memory location.
`xor` *`destmem`*`,` *`srcreg`*	*`destmem`* `:=` *`destmem`* XOR *`srcreg`* *`srcreg`* may be any 8-bit, 16-bit, 32-bit, or 64-bit general-purpose register; *`destmem`* can be any like-sized memory location.

Table A-106: EFLAGS Settings for `xor`

Flag	Setting
Carry	Cleared
Overflow	Cleared
Sign	Set if the logical XOR of the two operands has a `1` in the HO bit position.
Zero	Set if the logical XOR of the two operands produces a `0` result.