from

WRITE GREAT CODE

Volume 2: Thinking Low-Level, Writing High-Level

ONLINE APPENDIX A

The Minimal 80x86 Instruction Set

by Randall Hyde

NO STARCH
PRESS

San Francisco

For information on book distributors or translations, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.
555 De Haro Street, Suite 250, San Francisco, CA 94107
phone: 415.863.9900; fax: 415.863.9950; info@nostarch.com; www.nostarch.com

Library of Congress Cataloging-in-Publication Data (Volume 1)

```
Hyde, Randall.
  Write great code : understanding the machine / Randall Hyde.
       p. cm.
  ISBN 1-59327-003-8
1. Computer programming.  2. Computer architecture.  I. Title.
   QA76.6.H94 2004
   005.1--dc22
                                                    2003017502
```

If you haven't purchased a copy of *Write Great Code, Volume 2: Thinking Low-Level, Writing High-Level* and would like to do so, please go to www.nostarch.com.

A

THE MINIMAL 80X86 INSTRUCTION SET

Although the 80x86 CPU family supports hundreds of instructions, few compilers actually use more than a few dozen of these instructions. Many of the instructions have become obsolete over time because newer instructions have reduced the need for older instructions. Some instructions, such as the Pentium's MMX and SSE instructions, simply do not correspond to functions you'd normally perform in an HLL. Therefore, compilers rarely generate these types of machine instructions (such instructions generally appear only in handwritten assembly language programs). Therefore, you don't need to learn the entire 80x86 instruction set in order to study compiler output. Instead, you only need to learn the handful of instructions that the compiler actually emits on the 80x86. That's the purpose of this appendix, to describe those few instructions compilers actually use.

A.1 add

The add instruction requires two operands: a source operand and a destination operand. 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` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`add( constant, destmem );`	`destmem := destmem + constant` `destmem` must be an 8-bit, 16-bit, or 32-bit memory variable.
`add( srcreg, destreg );`	`destreg := destreg + srcreg` `destreg` and `srcreg` must be an 8-bit, 16-bit, or 32-bit general-purpose registers. They must both be the same size.
`add( srcmem, destreg );`	`destreg := destreg + srcmem` `destreg` must be an 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` must be an 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`$_n$ must be an 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`$_n$ must be an 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`$_{32}$	`destreg`$_n$:= `destreg`$_n$ + `srcreg`$_n$ `destreg`$_n$ and `srcreg`$_n$ must be an 8-bit, 16-bit, or 32-bit general-purpose registers, as specified by the suffix.
`addb srcmem`$_8$`, destreg`$_8$ `addw srcmem`$_{16}$`, destreg`$_{16}$ `addl srcmem`$_{32}$`, destreg`$_{32}$	`destreg`$_n$:= `destreg`$_n$ + `srcmem`$_n$ `destreg`$_n$ must be an 8-bit, 16-bit, or 32-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`$_{32}$	`destmem`$_n$:= `destmem`$_n$ + `srcreg`$_n$ `srcreg`$_n$ must be an 8-bit, 16-bit, or 32-bit general-purpose register, as specified by the suffix. `destmem`$_n$ can be any like-sized memory location.

Table A-3: MASM/TASM Syntax for add

Instruction	Description
`add destreg, constant`	`destreg := destreg + constant` `destreg` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`add destmem, constant`	`destmem := destmem + constant` `destmem` must be an 8-bit, 16-bit, or 32-bit memory variable.
`add destreg, srcreg`	`destreg := destreg + srcreg` `destreg` and `srcreg` must be an 8-bit, 16-bit, or 32-bit general-purpose registers. They must both be the same size.
`add destreg, srcmem`	`destreg := destreg + srcmem` `destreg` must be an 8-bit, 16-bit, or 32-bit general-purpose register; `srcmem` can be any like-sized memory location.
`add destmem, srcreg`	`destmem := destmem + srcreg` `srcreg` must be an 8-bit, 16-bit, or 32-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 one in its HO bit position.
Zero	Set if the sum of the two values is zero.

A.2 and

The and instruction requires two operands: a source operand and a destination operand. 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` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`and( constant, destmem );`	`destmem := destmem AND constant` `destmem` must be an 8-bit, 16-bit, or 32-bit memory variable.
`and( srcreg, destreg );`	`destreg := destreg AND srcreg` `destreg` and `srcreg` must be an 8-bit, 16-bit, or 32-bit general-purpose registers. They must both be the same size.

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

Instruction	Description
`and( srcmem, destreg );`	`destreg := destreg AND srcmem` `destreg` must be an 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` must be an 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`$_n$ must be an 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`$_n$ must be an 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`$_{32}$	`destreg`$_n$:= `destreg`$_n$ AND `srcreg`$_n$ `destreg`$_n$ and `srcreg`$_n$ must be an 8-bit, 16-bit, or 32-bit general-purpose registers, as specified by the suffix.
`andb srcmem`$_8$, `destreg`$_8$ `andw srcmem`$_{16}$, `destreg`$_{16}$ `andl srcmem`$_{32}$, `destreg`$_{32}$	`destreg`$_n$:= `destreg`$_n$ AND `srcmem`$_n$ `destreg`$_n$ must be an 8-bit, 16-bit, or 32-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`$_{32}$	`destmem`$_n$:= `destmem`$_n$ AND `srcreg`$_n$ `srcreg`$_n$ must be an 8-bit, 16-bit, or 32-bit general-purpose register, as specified by the suffix. `destmem`$_n$ can be any like-sized memory location.

Table A-7: MASM/TASM Syntax for `and`

Instruction	Description
`and destreg, constant`	`destreg := destreg AND constant` `destreg` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`and destmem, constant`	`destmem := destmem AND constant` `destmem` must be an 8-bit, 16-bit, or 32-bit memory variable.
`and destreg, srcreg`	`destreg := destreg AND srcreg` `destreg` and `srcreg` must be an 8-bit, 16-bit, or 32-bit general-purpose registers. They must both be the same size.
`and destreg, srcmem`	`destreg := destreg AND srcmem` `destreg` must be an 8-bit, 16-bit, or 32-bit general-purpose register. `srcmem` can be any like-sized memory location.
`and destmem, srcreg`	`destmem := destmem AND srcreg` `srcreg` must be an 8-bit, 16-bit, or 32-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 one in its HO bit position.
Zero	Set if the result is zero.

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). Next 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( reg32 ):`	Calls the subroutine at the address specified in the 32-bit register supplied as the single operand.
`call( mem32 );`	Calls the subroutine at the address held in the double-word memory location specified by the `mem32` 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 *reg32`	Calls the subroutine at the address specified in the 32-bit register supplied as the single operand.
`call *mem32`	Calls the subroutine at the address held in the double-word memory location specified by the `mem32` operand.

Table A-11: MASM/TASM Syntax for `call`

Instruction	Description
`call label`	Calls the subroutine that has the specified name (label) in the program.
`call reg32`	Calls the subroutine at the address specified in the 32-bit register supplied as the single operand.
`call mem32`	Calls the subroutine at the address held in the double-word memory location specified by the `mem32` operand.

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	Set the carry flag.

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

Instruction	Description
clc	Clears the carry flag.
cmc	Complements (inverts) the carry flag.
stc	Set 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 must be an 8-bit, 16-bit, or 32-bit general-purpose register.
cmp(mem, constant);	Compares mem against a constant. destmem must be an 8-bit, 16-bit, or 32-bit memory variable.
cmp(leftreg, rightreg);	Compares leftreg against rightreg. leftreg and rightreg must be an 8-bit, 16-bit, or 32-bit general-purpose registers. They must both be the same size.

Table A-15: HLA Syntax for `cmp` (continued)

Instruction	Description
`cmp( reg, mem);`	Compares a register with the value of a memory location. `reg` must be an 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` must be an 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, reg8`
`cmpw constant, reg16`
`cmpl constant, reg32`
	Compares reg_n against a constant.
	reg_n must be an 8-bit, 16-bit, or 32-bit general-purpose register, as appropriate for the suffix.

`cmpb constant, mem8`
`cmpw constant, mem16`
`cmpl constant, mem32`
	Compares mem_n against a constant.
	mem_n must be an 8-bit, 16-bit, or 32-bit memory variable, as appropriate for the suffix.

`cmpb leftreg8, rightreg8`
`cmpw leftreg16, rightreg16`
`cmpl leftreg32, rightreg32`
	Compares $rightreg_n$ to $leftreg_n$
	$rightreg_n$ and $leftreg_n$ must be an 8-bit, 16-bit, or 32-bit general-purpose registers, as specified by the suffix.

`cmpb mem8, reg8`
`cmpw mem16, reg16`
`cmpl mem32, reg32`
	Compares reg_n to mem_n
	reg_n must be an 8-bit, 16-bit, or 32-bit general-purpose register, according to the suffix. mem_n can be any like-sized memory location.

`cmpb reg8, mem8`
`cmpw reg16, mem16`
`cmpl reg32, mem32`
	Compares mem_n to reg_n
	reg_n must be an 8-bit, 16-bit, or 32-bit general-purpose register, as specified by the suffix. mem_n can be any like-sized memory location.

Table A-17: MASM/TASM Syntax for `cmp`

Instruction	Description
`cmp reg, constant`	Compares `reg` against a constant. `reg` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`cmp mem, constant`	Compares `mem` against a constant. `destmem` must be an 8-bit, 16-bit, or 32-bit memory variable.
`cmp leftreg, rightreg`	Compares `leftreg` against `rightreg`. `leftreg` and `rightreg` must be an 8-bit, 16-bit, or 32-bit general-purpose registers. They must both be the same size.
`cmp reg, mem`	Compares a register with the value of a memory location. `reg` must be an 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` must be an 8-bit, 16-bit, or 32-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	If the exclusive-OR of the overflow and sign flags is one after a comparison, then the first operand is less than the second operand when doing an unsigned comparison (for MASM/TASM and HLA, reverse the operands for Gas).
Sign	
Zero	Set if the two values are equal.

A.6 dec

The `dec` (decrement) instruction requires a single operand. The CPU subtracts one from this operand. This instruction also sets several flags in the EFLAGS register, based on the result, but you should 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` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`dec( mem );`	`mem := mem - 1` `mem` must be an 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}	reg_n `:=` reg_n `- 1` reg_n must be an 8-bit, 16-bit, or 32-bit general-purpose register, as appropriate for the suffix.
`decb` mem_8 `decw` mem_{16} `decl` mem_{32}	mem_n `:=` mem_n `- 1` mem_n must be an 8-bit, 16-bit, or 32-bit memory variable, as appropriate for the suffix.

Table A-21: MASM/TASM Syntax for `dec`

Instruction	Description
`dec reg`	`reg := reg - 1` `reg` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`dec mem`	`mem := mem - 1` `mem` must be an 8-bit, 16-bit, or 32-bit memory variable.

Table A-22: EFLAGS Settings for `dec`

Flag	Setting
Carry	Unaffected by the `dec` instruction.
Overflow	Set if subtracting one produces a signed underflow.

Table A-22: EFLAGS Settings for dec (continued)

Flag	Setting
Sign	Set if subtracting one produces a one in the HO bit position.
Zero	Set if subtracting one produces zero.

A.7 div

The div instruction takes a single operand. For an 8-bit operand (register or memory), the div instruction divides the 16-bit value in AX by that operand, producing the unsigned quotient in AL and the unsigned remainder in AH. For a 16-bit operand, the div instruction divides the 32-bit value in DX:AX (DX contains the HO word; AX contains the LO word), leaving the unsigned quotient in AX and the unsigned remainder in DX. For a 32-bit operand, the div instruction 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. This instruction scrambles the flags in the EFLAGS register; you cannot rely on their values after executing a div 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, or EAX (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 must be an 8-bit general-purpose register.
div(reg_{16});	ax := dx:ax div reg_{16} dx := dx:ax mod reg_{16} reg_{16} must be a 16-bit general-purpose register.
div(reg_{32});	eax := edx:eax div reg_{32} edx := edx:eax mod reg_{32} reg_{32} must be a 32-bit general-purpose register.
div(mem_8);	al := ax div mem_8 ah := ax mod mem_8 mem_8 must be an 8-bit memory location.
div(mem_{16});	ax := dx:ax div mem_{16} dx := dx:ax mod mem_{16} mem_{16} must be a 16-bit memory location.
div(mem_{32});	eax := edx:eax div mem_{32} edx := edx:eax mod mem_{32} mem_{32} must be a 32-bit memory location.

Table A-24: Gas Syntax for `div`

Instruction	Description

`divb reg8`	`al := ax div reg8`
	`ah := ax mod reg8`
	`reg8` must be an 8-bit general-purpose register.

`divw reg16`	`ax := dx:ax div reg16`
	`dx := dx:ax mod reg16`
	`reg16` must be a 16-bit general-purpose register.

`divl reg32`	`eax := edx:eax div reg32`
	`edx := edx:eax mod reg32`
	`reg32` must be a 32-bit general-purpose register.

`divb mem8`	`al := ax div mem8`
	`ah := ax mod mem8`
	`mem8` must be an 8-bit memory location.

`divw mem16`	`ax := dx:ax div mem16`
	`dx := dx:ax mod mem16`
	`mem16` must be a 16-bit memory location.

`divl mem32`	`eax := edx:eax div mem32`
	`edx := edx:eax mod mem32`
	`mem32` must be a 32-bit memory location.

Table A-25: MASM/TASM Syntax for `div`

Instruction	Description

`div reg8`	`al := ax div reg8`
	`ah := ax mod reg8`
	`reg8` must be an 8-bit general-purpose register.

`div reg16`	`ax := dx:ax div reg16`
	`dx := dx:ax mod reg16`
	`reg16` must be a 16-bit general-purpose register.

`div reg32`	`eax := edx:eax div reg32`
	`edx := edx:eax mod reg32`
	`reg32` must be a 32-bit general-purpose register.

`div mem8`	`al := ax div mem8`
	`ah := ax mod mem8`
	`mem8` must be an 8-bit memory location.

`div mem16`	`ax := dx:ax div mem16`
	`dx := dx:ax mod mem16`
	`mem16` must be a 16-bit memory location.

`div mem32`	`eax := edx:eax div mem32`
	`edx := edx:eax mod mem32`
	`mem32` must be a 32-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 Chapter 16 in *Write Great Code, Volume 2*) upon entry into a procedure. This instruction does the following:

1. It pushes the value of EBP onto the stack (see the discussion of the push instruction, later in this appendix).
2. It subtracts the value of its locals argument from the ESP register to allocate storage for local variables.
3. If the lexlevel argument is nonzero, the enter instruction builds a *display*. We will not discuss displays because you won't encounter them very often. For more details, see *The Art of Assembly Language* (No Starch Press, 2003). Most compilers, when they even use the enter instruction, specify zero 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 (see *The Art of Assembly Language* for details).
	Note: $locals_{16}$ is a 16-bit constant, and $lexlevel_8$ 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);
	Build display if $lexlevel_8$ is nonzero (see *The Art of Assembly Language* for details).
	Note: $locals_{16}$ is a 16-bit constant; $lexlevel_8$ is an 8-bit constant.

Table A-29: MASM/TASM Syntax for enter

Instruction	Description

enter $locals_{16}$, $lexlevel_8$	push(ebp);
	sub($locals_{16}$, ESP);
	Build display if $lexlevel_8$ is nonzero (see *The Art of Assembly Language* for details).
	Note: $locals_{16}$ is a 16-bit constant, and $lexlevel_8$ 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.

Table A-30: EFLAGS Settings for `enter` (continued)

Flag	Setting
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), then the `idiv` instruction 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, then the `idiv` instruction divides the 32-bit value in DX:AX (DX contains the HO word; 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, then the `idiv` instruction 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. This instruction scrambles the flags in the EFLAGS register; you cannot rely on their 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, or EAX (as appropriate).

Table A-31: HLA Syntax for `idiv`

Instruction	Description

`idiv( reg8 );`	`al := ax div reg8`
	`ah := ax mod reg8`
	`reg8` must be an 8-bit general-purpose register.

`idiv( reg16 );`	`ax := dx:ax div reg16`
	`dx := dx:ax mod reg16`
	`reg16` must be a 16-bit general-purpose register.

`idiv( reg32 );`	`eax := edx:eax div reg32`
	`edx := edx:eax mod reg32`
	`reg32` must be a 32-bit general-purpose register.

`idiv( mem8 );`	`al := ax div mem8`
	`ah := ax mod mem8`
	`mem8` must be an 8-bit memory location.

`idiv( mem16 );`	`ax := dx:ax div mem16`
	`dx := dx:ax mod mem16`
	`mem16` must be a 16-bit memory location.

`idiv( mem32 );`	`eax := edx:eax div mem32`
	`edx := edx:eax mod mem32`
	`mem32` must be a 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 must be an 8-bit general-purpose register.
`idivw` reg_{16}	`ax := dx:ax div` reg_{16} `dx := dx:ax mod` reg_{16} reg_{16} must be a 16-bit general-purpose register.
`idivl` reg_{32}	`eax := edx:eax div` reg_{32} `edx := edx:eax mod` reg_{32} reg_{32} must be a 32-bit general-purpose register.
`idivb` mem_8	`al := ax div` mem_8 `ah := ax mod` mem_8 mem_8 must be an 8-bit memory location.
`idivw` mem_{16}	`ax := dx:ax div` mem_{16} `dx := dx:ax mod` mem_{16} mem_{16} must be a 16-bit memory location.
`idivl` mem_{32}	`eax := edx:eax div` mem_{32} `edx := edx:eax mod` mem_{32} mem_{32} must be a 32-bit memory location.

Table A-33: MASM/TASM Syntax for `idiv`

Instruction	Description
`idiv` reg_8	`al := ax div` reg_8 `ah := ax mod` reg_8 reg_8 must be an 8-bit general-purpose register.
`idiv` reg_{16}	`ax := dx:ax div` reg_{16} `dx := dx:ax mod` reg_{16} reg_{16} must be a 16-bit general-purpose register.
`idiv` reg_{32}	`eax := edx:eax div` reg_{32} `edx := edx:eax mod` reg_{32} reg_{32} must be a 32-bit general-purpose register.
`idiv` mem_8	`al := ax div` mem_8 `ah := ax mod` mem_8 mem_8 must be an 8-bit memory location.
`idiv` mem_{16}	`ax := dx:ax div` mem_{16} `dx := dx:ax mod` mem_{16} mem_{16} must be a 16-bit memory location.
`idiv` mem_{32}	`eax := edx:eax div` mem_{32} `edx := edx:eax mod` mem_{32} mem_{32} must be a 32-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/TASM, and Gas, one form of this instruction has a single operand. If that operand is an 8-bit operand (register or memory), then the `imul` instruction multiplies the 8-bit value in AL by that operand, producing the signed product in AX. If that operand is a 16-bit operand, then the `imul` instruction 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 a 32-bit operand, then the `imul` instruction 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). This instruction scrambles the zero and sign flags in the EFLAGS register. You cannot rely on their 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/TASM continue to use the `imul` mnemonic for this instruction; 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( reg8 );`	`ax := al * reg8` `reg8` must be an 8-bit general-purpose register.
`imul( reg16 );`	`dx:ax := ax * reg16` `reg16` must be a 16-bit general-purpose register.
`imul( reg32 );`	`edx:eax := eax * reg32` `reg32` must be a 32-bit general-purpose register.
`imul( mem8 );`	`ax := al * mem8` `mem8` must be an 8-bit memory location.
`imul( mem16 );`	`dx:ax := ax * mem16` `mem16` must be a 16-bit memory location.
`imul( mem32 );`	`edx:eax := eax * mem32` `mem32` must be a 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 reg8`	`ax := al * reg8` `reg8` must be an 8-bit general-purpose register.
`imulw reg16`	`dx:ax := ax * reg16` `reg16` must be a 16-bit general-purpose register.
`imull reg32`	`edx:eax := eax * reg32` `reg32` must be a 32-bit general-purpose register.
`imulb mem8`	`ax := al * mem8` `mem8` must be an 8-bit memory location.
`imulw mem16`	`dx:ax := ax * mem16` `mem16` must be a 16-bit memory location.
`imull mem32`	`edx:eax := eax * mem32` `mem32` must be a 32-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.
`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.

Table A-37: MASM/TASM Syntax for `imul`

Instruction	Description
`imul reg8`	`ax := al * reg8` `reg8` must be an 8-bit general-purpose register.
`imul reg16`	`dx:ax := ax * reg16` `reg16` must be a 16-bit general-purpose register.
`imul reg32`	`edx:eax := eax * reg32` `reg32` must be a 32-bit general-purpose register.
`imul mem8`	`ax := al * mem8` `mem8` must be an 8-bit memory location.
`imul mem16`	`dx:ax := ax * mem16` `mem16` must be a 16-bit memory location.

Table A-37: MASM/TASM Syntax for `imul` (continued)

Instruction	Description
`imul mem32`	`edx:eax := eax * mem32` `mem32` must be a 32-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 or 32-bit general-purpose registers; they must both be the same size.
`imul destreg, mem`	`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-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 one to the value of this operand. This instruction also sets several flags in the EFLAGS register, based on the result. You should note, however, 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` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`inc( mem );`	`mem := mem + 1` `mem` must be an 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}	reg_n := reg_n + 1 reg_n must be an 8-bit, 16-bit, or 32-bit general-purpose register, as appropriate for the suffix.
`incb` mem_8 `incw` mem_{16} `incl` mem_{32}	mem_n := mem_n + 1 mem_n must be an 8-bit, 16-bit, or 32-bit memory variable, as appropriate for the suffix.

Table A-41: MASM/TASM Syntax for `inc`

Instruction	Description
`inc reg`	`reg := reg + 1` `reg` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`inc mem`	`mem := mem + 1` `mem` must be an 8-bit, 16-bit, or 32-bit memory variable.

Table A-42: EFLAGS Settings for `inc`

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

A.12 Conditional Jumps (Jcc)

The 80x86 supports a wide variety of conditional jumps 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 that the Jcc instructions do not affect any of the flags in the EFLAGS register. Here are the specific instructions and the conditions they test:

Table A-43: HLA, Gas, and MASM/TASM Conditional Jump Instructions

Instruction	Description

`ja label; //HLA`
`ja label ;Gas/MASM/TASM`
	Conditional jump if (unsigned) 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. Control falls through to the next instruction if the condition is false.

`jae label; //HLA`
`jae label ;Gas/MASM/TASM`
	Conditional jump if (unsigned) above or equal. See `ja` earlier in this table for details.

`jb label; //HLA`
`jb label ;Gas/MASM/TASM`
	Conditional jump if (unsigned) below. See `ja` earlier in this table for details.

`jbe label; //HLA`
`jbe label ;Gas/MASM/TASM`
	Conditional jump if (unsigned) below or equal. See `ja` earlier in this table for details.

Table A-43: HLA, Gas, and MASM/TASM Conditional Jump Instructions (continued)

Instruction	Description

`jc label; //HLA`
`jc label ;Gas/MASM/TASM`
	Conditional jump if carry is one. See `ja` earlier in this table for details.

`je label; //HLA`
`je label ;Gas/MASM/TASM`
	Conditional jump if equal. See `ja` earlier in this table for details.

`jg label; //HLA`
`jg label ;Gas/MASM/TASM`
	Conditional jump if (signed) greater. See `ja` earlier in this table for details.

`jge label; //HLA`
`jge label ;Gas/MASM/TASM`
	Conditional jump if (signed) greater or equal. See `ja` earlier in this table for details.

`jl label; //HLA`
`jl label ;Gas/MASM/TASM`
	Conditional jump if (signed) less than. See `ja` earlier in this table for details.

`jle label; //HLA`
`jle label ;Gas/MASM/TASM`
	Conditional jump if (signed) less than or equal. See `ja` earlier in this table for details.

`jna label; //HLA`
`jna label ;Gas/MASM/TASM`
	Conditional jump if (unsigned) not above. See `ja` earlier in this table for details.

`jnae label; //HLA`
`jnae label ;Gas/MASM/TASM`
	Conditional jump if (unsigned) not above or equal. See `ja` earlier in this table for details.

`jnb label; //HLA`
`jnb label ;Gas/MASM/TASM`
	Conditional jump if (unsigned) below. See `ja` earlier in this table for details.

`jnbe label; //HLA`
`jnbe label ;Gas/MASM/TASM`
	Conditional jump if (unsigned) below or equal. See `ja` earlier in this table for details.

`jnc label; //HLA`
`jnc label ;Gas/MASM/TASM`
	Conditional jump if carry flag is clear (no carry). See `ja` earlier in this table for details.

`jne label; //HLA`
`jne label ;Gas/MASM/TASM`
	Conditional jump if not equal. See `ja` earlier in this table for details.

`jng label; //HLA`
`jng label ;Gas/MASM/TASM`
	Conditional jump if (signed) not greater. See `ja` earlier in this table for details.

`jnge label; //HLA`
`jnge label ;Gas/MASM/TASM`
	Conditional jump if (signed) not greater or equal. See `ja` earlier in this table for details.

`jnl label; //HLA`
`jnl label ;Gas/MASM/TASM`
	Conditional jump if (signed) not less than. See `ja` earlier in this table for details.

`jnle label; //HLA`
`jnle label ;Gas/MASM/TASM`
	Conditional jump if (signed) not less than or equal. See `ja` earlier in this table for details.

`jno label; //HLA`
`jno label ;Gas/MASM/TASM`
	Conditional jump if no overflow (overflow flag = 0). See `ja` earlier in this table for details.

`jns label; //HLA`
`jns label ;Gas/MASM/TASM`
	Conditional jump if no sign (sign flag = 0). See `ja` earlier in this table for details.

`jnz label; //HLA`
`jnz label ;Gas/MASM/TASM`
	Conditional jump if not zero (zero flag = 0). See `ja` earlier in this table for details.

`jo label; //HLA`
`jo label ;Gas/MASM/TASM`
	Conditional jump if overflow (overflow flag = 1). See `ja` earlier in this table for details.

`js label; //HLA`
`js label ;Gas/MASM/TASM`
	Conditional jump if sign (sign flag = 1). See `ja` earlier in this table for details.

`jz label; //HLA`
`jz label ;Gas/MASM/TASM`
	Conditional jump if zero (zero flag = 1). See `ja` earlier in this table for details.

Table A-43: HLA, Gas, and MASM/TASM Conditional Jump Instructions (continued)

Instruction	Description

`jcxz` *`label`*`; // HLA syntax`
`jcxz label ;Gas/MASM/TASM`
	Conditional jump if CX is zero. See `ja` earlier in this table for details. Note: The range of this branch is limited to ±128 bytes around the instruction.

`jecxz` *`label`*`; // HLA syntax`
`jecxz label ; MASM/TASM/Gas`
	Conditional jump if ECX is zero. See `ja` earlier in this table for details. Note: The range of this branch is limited to ±128 bytes around the instruction.

A.13 jmp

The `jmp` instruction unconditionally transfers control ("jumps") 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`$_{32}$	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-46: MASM/TASM Syntax for `jmp`

Instruction	Description
`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`$_{32}$.

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 that loads a register with the contents of a memory location. Like mov, the lea instruction does not affect any bits 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 copy 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}	reg_{32} := address of mem reg_{32} must be a 32-bit general-purpose register. mem can be any sized memory location.

Table A-49: MASM/TASM Syntax for lea

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

A.15 leave

The leave instruction cleans up after a procedure, removing local variable storage and restoring the EBP register to its original value.

1. It copies the value of EBP into ESP.
2. It pops EBP's value from the stack.

Table A-50: Gas/MASM/TASM Syntax for leave

Instruction	Description

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

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 operand and a destination operand. It copies the value from the source operand to the destination operand. It does not affect any bits in the EFLAGS register.

Table A-52: HLA Syntax for `mov`

Instruction	Description
`mov( constant, destreg );`	`destreg := constant` `destreg` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`mov( constant, destmem );`	`destmem := constant` `destmem` must be an 8-bit, 16-bit, or 32-bit memory variable.
`mov( srcreg, destreg );`	`destreg := srcreg` `destreg` and `srcreg` must be 8-bit, 16-bit, or 32-bit general-purpose registers. They must both be the same size.
`mov( srcmem, destreg );`	`destreg := srcmem` `destreg` must be an 8-bit, 16-bit, or 32-bit general-purpose register. `srcmem` can be any like-sized memory location.
`mov( srcreg, destmem );`	`destmem := srcreg` `srcreg` must be an 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`$_n$ must be an 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`$_n$ must be an 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`$_{32}$	`destreg`$_n$ `:= srcreg`$_n$ `destreg`$_n$ and `srcreg`$_n$ must be 8-bit, 16-bit, or 32-bit general-purpose registers, as specified by the suffix.

Table A-53: Gas Syntax for `mov` (continued)

Instruction	Description

`movb srcmem8, destreg8`
`movw srcmem16, destreg16`
`movl srcmem32, destreg32`
	`destregn :=srcmemn`
	`destregn` must be an 8-bit, 16-bit, or 32-bit general-purpose register; according to the suffix, `srcmemn` can be any like-sized memory location.

`movb srcreg8, destmem8`
`movw srcreg16, destmem16`
`movl srcreg32, destmem32`
	`destmemn := srcregn`
	`srcregn` must be an 8-bit, 16-bit, or 32-bit general-purpose register; as specified by the suffix, `destmemn` can be any like-sized memory location.

Table A-54: MASM/TASM Syntax for `mov`

Instruction	Description
`mov destreg, constant`	`destreg := constant` `destreg` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`mov destmem, constant`	`destmem := constant` destmem must be an 8-bit, 16-bit, or 32-bit memory variable.
`mov destreg, srcreg`	`destreg := srcreg` `destreg` and `srcreg` must be 8-bit, 16-bit, or 32-bit general-purpose registers. They must both be the same size.
`mov destreg, srcmem`	`destreg := srcmem` `destreg` must be an 8-bit, 16-bit, or 32-bit general-purpose register. `srcmem` can be any like-sized memory location.
`mov destmem, srcreg`	`destmem := srcreg` `srcreg` must be an 8-bit, 16-bit, or 32-bit general-purpose register. `destmem` can be any like-sized memory location.

A.17 movs, movsb, movsd, movsw

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

Without a repeat prefix, these instructions copy a byte (`movsb`), word (`movsw`), or double word (`movsd`) 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 one, `movsw` increments or decrements them by two, and `movsd` increments or decrements them by four. 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 clear, the move string instructions decrement ESI and EDI.

If the repeat prefix is attached to one of these move string instructions, then 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`, `movsw`

Instruction	Description

`movsb`
`movsw`
`movsl`
	`[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 movsl`
	`destreg := srcmem`
	Zero extends the value of `srcmem` to the size of `destreg`.
	`destreg` must be a 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-57: MASM/TASM Syntax for `movsb`, `movsd`, `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`
	`destreg := srcmem`
	Zero extends the value of `srcmem` to the size of `destreg`.
	`destreg` must be a 16-bit or 32-bit general-purpose register. `srcmem` is an 8-bit or 16-bit memory location that is smaller than `destreg`.

A.18 movsx, movzx

The `movsx` and `movzx` instructions require two operands: a source operand and a destination operand. 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`). See *Write Great Code,*

Volume 1 for details on these operations. Compilers use these instructions to translate smaller values to larger objects. These instructions do not affect any flags.

Table A-58: HLA Syntax for `movsx`, `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 that is 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 that is smaller than `destreg`.
`movsx( srcmem, destreg );`	`destreg := srcmem` Sign extends the value of `srcmem` to the size of `destreg`. `destreg` must be a 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` must be a 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 `movsx`, `movzx` (`movsbw`, `movsbl`, `movswl`, `movzbw`, `movzbl`, and `movzwl`)

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

Table A-60: MASM/TASM Syntax for `movsx`, `movzx`

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

A.19 mul

The `mul` instruction allows a single operand. If that operand is an 8-bit operand (register or memory), then the `mul` instruction multiplies the 8-bit value in AL by that operand, producing an unsigned product in AX. If that operand is a 16-bit operand, then the `mul` instruction multiplies the 16-bit value in AX by the operand, leaving the unsigned product in DX:AX (DX contains the HO word; AX contains the LO word). If the operand is a 32-bit operand, then the `mul` instruction 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.) This instruction scrambles the zero and sign bits in the EFLAGS register; you cannot rely on their values after executing an `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( reg8 );`	`ax := al * reg8` `reg8` must be an 8-bit general-purpose register.
`mul( reg16 );`	`dx:ax := ax * reg16` `reg16` must be a 16-bit general-purpose register.
`mul( reg32 );`	`edx:eax := eax * reg32` `reg32` must be a 32-bit general-purpose register.
`mul( mem8 );`	`ax := al * mem8` `mem8` must be an 8-bit memory location.
`mul( mem16 );`	`dx:ax := ax * mem16` `mem16` must be a 16-bit memory location.
`mul( mem32 );`	`edx:eax := eax * mem32` `mem32` must be a 32-bit memory location.

Table A-62: Gas Syntax for `mul`

Instruction	Description
`mulb` reg_8	`ax := al *` reg_8 reg_8 must be an 8-bit general-purpose register.
`mulw` reg_{16}	`dx:ax := ax *` reg_{16} reg_{16} must be a 16-bit general-purpose register.
`mull` reg_{32}	`edx:eax := eax *` reg_{32} reg_{32} must be a 32-bit general-purpose register.
`mulb` mem_8	`ax := al *` mem_8 mem_8 must be an 8-bit memory location.
`mulw` mem_{16}	`dx:ax := ax *` mem_{16} mem_{16} must be a 16-bit memory location.
`mull` mem_{32}	`edx:eax := eax *` mem_{32} mem_{32} must be a 32-bit memory location.

Table A-63: MASM/TASM Syntax for `mul`

Instruction	Description
`mul` reg_8	`ax := al *` reg_8 reg_8 must be an 8-bit general-purpose register.
`mul` reg_{16}	`dx:ax := ax *` reg_{16} reg_{16} must be a 16-bit general-purpose register.
`mul` reg_{32}	`edx:eax := eax *` reg_{32} reg_{32} must be a 32-bit general-purpose register.
`mul` mem_8	`ax := al *` mem_8 mem_8 must be an 8-bit memory location.
`mul` mem_{16}	`dx:ax := ax *` mem_{16} mem_{16} must be a 16-bit memory location.
`mul` mem_{32}	`edx:eax := eax *` mem_{32} mem_{32} must be a 32-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` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`neg( mem );`	`mem := - mem` `mem` must be an 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}	reg_n := -reg_n reg_n must be an 8-bit, 16-bit, or 32-bit general-purpose register, as appropriate for the suffix.
`negb` mem_8 `negw` mem_{16} `negl` mem_{32}	mem_n := -mem_n mem_n must be an 8-bit, 16-bit, or 32-bit memory variable, as appropriate for the suffix.

Table A-67: MASM/TASM Syntax for neg

Instruction	Description
`neg reg`	`reg := -reg` `reg` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`neg mem`	`mem := -mem` `mem` must be an 8-bit, 16-bit, or 32-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 that can fit in the size specified (which cannot be negated in the two's complement system). Example: With a byte operand, if you negate –128, the result (+128) no longer fits in a byte. The largest number that fits in a byte is +127.
Sign	Set if negation produces a one in the HO bit position.
Zero	Set if negation produces zero (i.e., the value was originally zero).

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` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`not( mem );`	`mem := not mem` `mem` must be an 8-bit, 16-bit, or 32-bit memory variable.

Table A-70: Gas Syntax for `not`

Instruction	Description

`notb reg8`
`notw reg16`
`notl reg32`
	reg_n := not reg_n
	reg_n must be an 8-bit, 16-bit, or 32-bit general-purpose register, as appropriate for the suffix.

`notb mem8`
`notw mem16`
`notl mem32`
	mem_n := not mem_n
	mem_n must be an 8-bit, 16-bit, or 32-bit memory variable, as appropriate for the suffix.

Table A-71: MASM/TASM Syntax for `not`

Instruction	Description
`not reg`	`reg := not reg` `reg` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`not mem`	`mem := not mem` `mem` must be an 8-bit, 16-bit, or 32-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 one in the HO bit position.
Zero	Set if logical NOT produces zero (i.e., the value was originally all one bits).

A.22 or

The `or` instruction requires two operands: a source operand and a destination operand. 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` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`or( constant, destmem );`	`destmem := destmem OR constant` `destmem` must be an 8-bit, 16-bit, or 32-bit memory variable.

Table A-73: HLA Syntax for `or` (continued)

Instruction	Description
`or( srcreg, destreg );`	`destreg := destreg OR srcreg` `destreg` and `srcreg` must be 8-bit, 16-bit, or 32-bit general-purpose registers. They must both be the same size.
`or( srcmem, destreg );`	`destreg := destreg OR srcmem` `destreg` must be an 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` must be an 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`$_n$ must be an 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`$_n$ must be an 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`$_{32}$	`destreg`$_n$ `:= destreg`$_n$ `OR srcreg`$_n$ `destreg`$_n$ and `srcreg`$_n$ must be 8-bit, 16-bit, or 32-bit general-purpose registers, as specified by the suffix.
`orb srcmem`$_8$`, destreg`$_8$ `orw srcmem`$_{16}$`, destreg`$_{16}$ `orl srcmem`$_{32}$`, destreg`$_{32}$	`destreg`$_n$ `:= destreg`$_n$ `OR srcmem`$_n$ `destreg`$_n$ must be an 8-bit, 16-bit, or 32-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`$_{32}$	`destmem`$_n$ `:= destmem`$_n$ `OR srcreg`$_n$ `srcreg`$_n$ must be an 8-bit, 16-bit, or 32-bit general-purpose register, as specified by the suffix; `destmem`$_n$ can be any like-sized memory location.

Table A-75: MASM/TASM Syntax for `or`

Instruction	Description
`or destreg, constant`	`destreg := destreg OR constant` `destreg` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`or destmem, constant`	`destmem := destmem OR constant` `destmem` must be an 8-bit, 16-bit, or 32-bit memory variable.
`or destreg, srcreg`	`destreg := destreg OR srcreg` `destreg` and `srcreg` must be 8-bit, 16-bit, or 32-bit general-purpose registers. They must both be the same size.
`or destreg, srcmem`	`destreg := destreg OR srcmem` `destreg` must be an 8-bit, 16-bit, or 32-bit general-purpose register. `srcmem` can be any like-sized memory location.
`or destmem, srcreg`	`destmem := destmem OR srcreg` `srcreg` must be an 8-bit, 16-bit, or 32-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 one in its HO bit position.
Zero	Set if the result is zero.

A.23 push, pushfd, pushd, and pushw

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

1. Subtract the size of the operand in bytes (2 or 4) from the ESP.
2. Store a copy of the operand's value at the memory location now referenced by ESP.

The `pushfd` instruction pushes a copy of the 80x86 EFLAGS register onto the stack (four bytes). Often, you will see the instructions `pushd` and `pushw` in some 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 constant 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 `push`, `pushfd`, `pushw`, and `pushd`

Instruction	Description
`pushw constant`	Pushes a 16-bit constant onto the stack.
`pushl constant`	Pushes a 32-bit constant onto the stack.
`pushw` $srcreg_{16}$ `pushl` $srcreg_{32}$	Pushes a register onto the stack. $srcreg_n$ must be a 16-bit or 32-bit general-purpose register, as appropriate for the instruction.

Table A-78: Gas Syntax for `push`, `pushfd`, `pushw`, and `pushd` (continued)

Instruction	Description
`pushw srcmem16` `pushl srcmem32`	Pushes the contents of a memory location onto the stack. `srcmem`$_n$ must be a 16-bit or 32-bit memory variable, as appropriate for the instruction.
`pushfd`	Pushes a copy of the EFLAGS register onto the stack.

Table A-79: MASM/TASM Syntax for `push`, `pushfd`, `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 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.

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). The `pop` instruction requires a single 16-bit or 32-bit register or memory operand, and it 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 and move this data to the location specified by the operand.
2. Add the size of the operand in bytes (2 or 4) to the ESP 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, the `popfd` instruction replaces all the allowed flags with the value read from the stack. Please note that depending on the privilege level of the current task, the CPU will only allow some of the flags to be modified on the copy of the EFLAGS register.

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`$_{32}$	Pops a value from the stack into a register. `destreg`$_n$ must be a 16-bit or 32-bit general-purpose register, as appropriate for the instruction suffix.
`popw destmem`$_{16}$ `popl destmem`$_{32}$	Pops a value from the stack into a memory variable. `destmem`$_n$ must be a 16-bit or 32-bit memory variable, as appropriate for the instruction.
`popfd`	Pushes a copy of the EFLAGS register onto the stack.

Table A-82: MASM/TASM 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-83: EFLAGS Settings for `popfd`

Flag	Setting
Carry	Set or cleared according to the value found on the stack.
Overflow	Set or cleared according to the value found on the stack.
Sign	Set or cleared according to the value found on the stack.
Zero	Set or cleared according to the value found on 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 a return 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 bits and adds it to the ESP 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 register, and then transfers control to the return address.

Table A-86: MASM/TASM 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.

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, shifting them 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 in the EFLAGS register. The HO bit is unaffected by the `sar` instruction.

The `shr` (shift right, or shift logical right) instruction shifts all the bits in the destination operand from HO bit positions to LO bit positions, shifting them 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 in the EFLAGS register. This instruction shifts a zero into the HO bit position after each bit shift occurs.

The `shl` (shift left, or shift logical left) instruction shifts all the bits in the destination operand from LO bit positions to HO bit positions shifting them 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 in the EFLAGS register. This instruction shifts a zero 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` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`shl( constant, destmem );`	`destmem := destmem SHL constant` `destmem` must be an 8-bit, 16-bit, or 32-bit memory variable.
`shl( cl, destreg );`	`destreg := destreg SHL CL` `destreg` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`shl( cl, destmem );`	`destmem := destmem SHL CL` `destmem` must be an 8-bit, 16-bit, or 32-bit memory variable.
`shr( constant, destreg );`	`destreg := destreg SHR constant` `destreg` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`shr( constant, destmem );`	`destmem := destmem SHR constant` `destmem` must be an 8-bit, 16-bit, or 32-bit memory variable.
`shr( cl, destreg );`	`destreg := destreg SHR CL` `destreg` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`shr( cl, destmem );`	`destmem := destmem SHR CL` `destmem` must be an 8-bit, 16-bit, or 32-bit memory variable.
`sar( constant, destreg );`	`destreg := destreg SAR constant` `destreg` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`sar( constant, destmem );`	`destmem := destmem SAR constant` `destmem` must be an 8-bit, 16-bit, or 32-bit memory variable.
`sar( cl, destreg );`	`destreg := destreg SAR CL` `destreg` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`sar( cl, destmem );`	`destmem := destmem SAR CL` `destmem` must be an 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`$_{32}$	`destreg`$_n$ `:= destreg`$_n$ `SHL constant` `destreg`$_n$ must be an 8-bit, 16-bit, or 32-bit general-purpose register, as appropriate for the instruction.
`shlb constant, destmem`$_8$ `shlw constant, destmem`$_{16}$ `shll constant, destmem`$_{32}$	`destmem`$_n$ `:= destmem`$_n$ `SHL constant` `destmem`$_n$ must be an 8-bit, 16-bit, or 32-bit memory variable, as appropriate for the instruction.
`shlb cl, destreg`$_8$ `shlw cl, destreg`$_{16}$ `shll cl, destreg`$_{32}$	`destreg`$_n$ `:= destreg`$_n$ `SHL CL` `destreg`$_n$ must be an 8-bit, 16-bit, or 32-bit general-purpose register, as appropriate for the instruction.
`shlb cl, destmem`$_8$ `shlw cl, destmem`$_{16}$ `shll cl, destmem`$_{32}$	`destmem`$_n$ `:= destmem`$_n$ `SHL CL` `destmem`$_n$ must be an 8-bit, 16-bit, or 32-bit memory variable, as appropriate for the instruction.

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

Instruction	Description
shrb constant, $destreg_8$ shrw constant, $destreg_{16}$ shrl constant, $destreg_{32}$	$destreg_n$:= $destreg_n$ SHR constant $destreg_n$ must be an 8-bit, 16-bit, or 32-bit general-purpose register, as appropriate for the instruction.
shrb constant, $destmem_8$ shrw constant, $destmem_{16}$ shrl constant, $destmem_{32}$	$destmem_n$:= $destmem_n$ SHR constant $destmem_n$ must be an 8-bit, 16-bit, or 32-bit memory variable, as appropriate for the instruction.
shrb cl, $destreg_8$ shrw cl, $destreg_{16}$ shrl cl, $destreg_{32}$	$destreg_n$:= $destreg_n$ SHR CL $destreg_n$ must be an 8-bit, 16-bit, or 32-bit general-purpose register, as appropriate for the instruction.
shrb cl, $destmem_8$ shrw cl, $destmem_{16}$ shrl cl, $destmem_{32}$	$destmem_n$:= $destmem_n$ SHR CL $destmem_n$ must be an 8-bit, 16-bit, or 32-bit memory variable, as appropriate for the instruction.
sarb constant, $destreg_8$ sarw constant, $destreg_{16}$ sarl constant, $destreg_{32}$	$destreg_n$:= $destreg_n$ SAR constant $destreg_n$ must be an 8-bit, 16-bit, or 32-bit general-purpose register, as appropriate for the instruction.
sarb constant, $destmem_8$ sarw constant, $destmem_{16}$ sarl constant, $destmem_{32}$	$destmem_n$:= $destmem_n$ SAR constant $destmem_n$ must be an 8-bit, 16-bit, or 32-bit memory variable, as appropriate for the instruction.
sarb cl, $destreg_8$ sarw cl, $destreg_{16}$ sarl cl, $destreg_{32}$	$destreg_n$:= $destreg_n$ SAR CL $destreg_n$ must be an 8-bit, 16-bit, or 32-bit general-purpose register, as appropriate for the instruction.
sarb cl, $destmem_8$ sarlw cl, $destmem_{16}$ sarl cl, $destmem_{32}$	$destmem_n$:= $destmem_n$ SAR CL $destmem_n$ must be an 8-bit, 16-bit, or 32-bit memory variable, as appropriate for the instruction.

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

Instruction	Description
shl destreg, constant	destreg := destreg SHL constant destreg must be an 8-bit, 16-bit, or 32-bit general-purpose register.
shl destmem, constant	destmem := destmem SHL constant destmem must be an 8-bit, 16-bit, or 32-bit memory variable.
shl destreg, cl	destreg := destreg SHL CL destreg must be an 8-bit, 16-bit, or 32-bit general-purpose register.
shl destmem, cl	destmem := destmem SHL CL destmem must be an 8-bit, 16-bit, or 32-bit memory variable.
shr destreg, constant	destreg := destreg SHR constant destreg must be an 8-bit, 16-bit, or 32-bit general-purpose register.
shr destmem, constant	destmem := destmem SHR constant destmem must be an 8-bit, 16-bit, or 32-bit memory variable.
shr destreg, cl	destreg := destreg SHR CL destreg must be an 8-bit, 16-bit, or 32-bit general-purpose register.
shr destmem, cl	destmem := destmem SHR CL destmem must be an 8-bit, 16-bit, or 32-bit memory variable.
sar destreg, constant	destreg := destreg SAR constant destreg must be an 8-bit, 16-bit, or 32-bit general-purpose register.
sar destmem, constant	destmem := destmem SAR constant destmem must be an 8-bit, 16-bit, or 32-bit memory variable.

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

Instruction	Description
`sar destreg, cl`	`destreg := destreg SAR CL` `destreg` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`sar destmem, cl`	`destmem := destmem SAR CL` `destmem` must be an 8-bit, 16-bit, or 32-bit memory variable.

Table A-90: EFLAGS Settings for `shl`, `sar`, `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 one in its HO bit position.
Zero	Set if the result is zero.

* Actually, the flags are only defined if the shift count is one.

A.27 Conditional Set (Scc) Instructions

The 80x86 supports a wide variety of conditional set instructions that set an 8-bit register or memory location to zero or one based upon tests on the EFLAGS register. These instructions allow the CPU to set Boolean variables to true or false based on conditions computed by instructions such as `cmp` that affect the EFLAGS register. Note that these instructions do not modify the EFLAGS register. Here are the specific instructions and the conditions they test:

Table A-91: Conditional Set Instructions

Instruction	Description

HLA:
`seta( reg8 );`
`seta( mem8 );`
MASM/TASM/Gas:
`seta reg8`
`seta mem8`
	Conditional set if (unsigned) above (carry = 0 and zero = 0). Stores a one 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 zero into the destination operand otherwise.

HLA:
`setae( reg8 );`
`setae( mem8 );`
MASM/TASM/Gas:
`setae reg8`
`setae mem8`
	Conditional set if (unsigned) above or equal (carry = 0). See the description for `seta` for details.

HLA:
`setb( reg8 );`
`setb( mem8 );`
MASM/TASM/Gas:
`setb reg8`
`setb mem8`
	Conditional set if (unsigned) below (carry = 1). See the description for `seta` for details.

Table A-91: Conditional Set Instructions (continued)

Instruction	Description

HLA:
`setbe( reg8 );`
`setbe( mem8 );`
MASM/TASM/Gas:
`setbe reg8`
`setbe mem8`
	Conditional set if (unsigned) below or equal (carry = 1 or zero = 1). See the description for seta for details.

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

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

HLA:
`setg( reg8 );`
`setg( mem8 );`
MASM/TASM/Gas:
`setg reg8`
`setg mem8`
	Conditional set if (signed) greater (sign = overflow and zero = 0). See the description for seta for details.

HLA:
`setge( reg8 );`
`setge( mem8 );`
MASM/TASM/Gas:
`setge reg8`
`setge mem8`
	Conditional set if (signed) greater or equal (sign = overflow or zero = 1). See the description for seta for details.

HLA:
`setl( reg8 );`
`setl( mem8 );`
MASM/TASM/Gas:
`setl reg8`
`setl mem8`
	Conditional set if (signed) less than (sign <> overflow). See the description for seta for details.

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

HLA:
`setna( reg8 );`
`setna( mem8 );`
MASM/TASM/Gas:
`setna reg8`
`setna mem8`
	Conditional set if (unsigned) not above (carry = 1 or zero = 1). See the description for seta for details.

Table A-91: Conditional Set Instructions (continued)

Instruction	Description

HLA:
`setnae( reg8 );`
`setnae( mem8 );`
MASM/TASM/Gas:
`setnae reg8`
`setnae mem8`
	Conditional set if (unsigned) not above or equal (carry = 1). See the description for `seta` for details.

HLA:
`setnb( reg8 );`
`setnb( mem8 );`
MASM/TASM/Gas:
`setnb reg8`
`setnb mem8`
	Conditional set if (unsigned) not below (carry = 0). See the description for `seta` for details.

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

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

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

HLA:
`setng( reg8 );`
`setng( mem8 );`
MASM/TASM/Gas:
`setng reg8`
`setng mem8`
	Conditional set if (signed) not greater (sign <> overflow or zero = 1). See the description for `seta` for details.

HLA:
`setnge( reg8 );`
`setnge( mem8 );`
MASM/TASM/Gas:
`setnge reg8`
`setnge mem8`
	Conditional set if (signed) not greater than (sign <> overflow). See the description for `seta` for details.

HLA:
`setnl( reg8 );`
`setnl( mem8 );`
MASM/TASM/Gas:
`setnl reg8`
`setnl mem8`
	Conditional set if (signed) not less than (sign = overflow). See the description for `seta` for details.

Table A-91: Conditional Set Instructions (continued)

Instruction	Description

HLA:
`setnle( reg8 );`
`setnle( mem8 );`
MASM/TASM/Gas:
`setnle reg8`
`setnle mem8`
	Conditional set if (signed) not less than or equal (sign = overflow and zero = 0). See the description for seta for details.

HLA:
`setno( reg8 );`
`setno( mem8 );`
MASM/TASM/Gas:
`setno reg8`
`setno mem8`
	Conditional set if no overflow (overflow = 0). See the description for seta for details.

HLA:
`setns( reg8 );`
`setns( mem8 );`
MASM/TASM/Gas:
`setns reg8`
`setns mem8`
	Conditional set if no sign (sign = 0). See the description for seta for details.

HLA:
`setnz( reg8 );`
`setnz( mem8 );`
MASM/TASM/Gas:
`setnz reg8`
`setnz mem8`
	Conditional set if not zero (zero = 0). See the description for seta for details.

HLA:
`seto( reg8 );`
`seto( mem8 );`
MASM/TASM/Gas:
`seto reg8`
`seto mem8`
	Conditional set if overflow (overflow = 1). See the description for seta for details.

HLA:
`sets( reg8 );`
`sets( mem8);`
MASM/TASM/Gas:
`sets reg8`
`sets mem8`
	Conditional set if sign set (sign = 1). See the description for seta for details.

HLA:
`setz( reg8 );`
`setz( mem8 );`
MASM/TASM/Gas:
`setz reg8`
`setz mem8`
	Conditional set if zero (zero = 1). See the description for seta for details.

A.28 stos, stosb, stosd, stosw

The stos instructions do not require any explicit operands. These are the *store string* instructions that copy a value in AL, AX, or EAX into a range of memory locations. 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`), or EAX (`stosd`) to the memory location pointed at by EDI (the destination index register). After copying the data, the CPU either increments or decrements EDI by the size, in bytes, of the transfer. That is, `stosb` increments or decrements EDI by one, `stosw` increments or decrements EDI by two, and `stosd` increments or decrements EDI by four. These instructions determine whether to increment or decrement EDI based on the value of the *direction flag* in the EFLAGs register. If the direction flag is clear, the store string instruction increments EDI; if the direction flag is clear, the store string instruction decrements EDI.

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 register. Compilers typically use this instruction to clear out a block of bytes in memory (that is, set the block of bytes to all zeros).

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`, `stosw`

Instruction	Description

`stosb`
`stosw`
`stosl`
	`[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 movsb`
`rep movsw`
`rep movsd`
	`[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-94: MASM/TASM Syntax for stosb, stosd, 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 movsb rep movsw rep movsd	[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.

A.29 sub

The sub instruction requires two operands: a source operand and a destination operand. 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 must be an 8-bit, 16-bit, or 32-bit general-purpose register.
sub(constant, destmem);	destmem := destmem - constant destmem must be an 8-bit, 16-bit, or 32-bit memory variable.
sub(srcreg, destreg);	destreg := destreg - srcreg destreg and srcreg must be an 8-bit, 16-bit, or 32-bit general-purpose registers. They must both be the same size.
sub(srcmem, destreg);	destreg := destreg - srcmem destreg must be an 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 must be an 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$_n$ must be an 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$_n$ must be an 8-bit, 16-bit, or 32-bit memory variable, as appropriate for the suffix.

subb srcreg$_8$, destreg$_8$
subw srcreg$_{16}$, destreg$_{16}$
subl srcreg$_{32}$, destreg$_{32}$
	destreg$_n$:= destreg$_n$ - srcreg$_n$
	destreg$_n$ and srcreg$_n$ must be 8-bit, 16-bit, or 32-bit general-purpose registers, as specified by the suffix.

subb srcmem$_8$, destreg$_8$
subw srcmem$_{16}$, destreg$_{16}$
subl srcmem$_{32}$, destreg$_{32}$
	destreg$_n$:= destreg$_n$ - srcmem$_n$
	destreg$_n$ must be an 8-bit, 16-bit, or 32-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$_{32}$
	destmem$_n$:= destmem$_n$ - srcreg$_n$
	srcreg$_n$ must be an 8-bit, 16-bit, or 32-bit general-purpose register, as specified by the suffix; destmem$_n$ can be any like-sized memory location.

Table A-97: MASM/TASM Syntax for sub

Instruction	Description
sub destreg, constant	destreg := destreg - constant destreg must be an 8-bit, 16-bit, or 32-bit general-purpose register.
sub destmem, constant	destmem := destmem - constant destmem must be an 8-bit, 16-bit, or 32-bit memory variable.
sub destreg, srcreg	destreg := destreg - srcreg destreg and srcreg must be 8-bit, 16-bit, or 32-bit general-purpose registers. They must both be the same size.
sub destreg, srcmem	destreg := destreg - srcmem destreg must be an 8-bit, 16-bit, or 32-bit general-purpose register. srcmem can be any like-sized memory location.
sub destmem, srcreg	destmem := destmem - srcreg srcreg must be an 8-bit, 16-bit, or 32-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 one in its HO bit position.
Zero	Set if the difference of the two values is zero.

A.30 test

The test instruction requires two operands: a source operand and a destination operand. 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 zero (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` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`test( constant, destmem );`	`destmem - constant (result to EFLAGS)` `destmem` must be an 8-bit, 16-bit, or 32-bit memory variable.
`test( srcreg, destreg );`	`destreg - srcreg (result to EFLAGS)` `destreg` and `srcreg` must be 8-bit, 16-bit, or 32-bit general-purpose registers. They must both be the same size.
`test( srcmem, destreg );`	`destreg - srcmem (result to EFLAGS)` `destreg` must be an 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` must be an 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`$_n$ must be an 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`$_n$ must be an 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`$_{32}$	`destreg`$_n$ `- srcreg`$_n$ `(result to EFLAGS)` `destreg`$_n$ and `srcreg`$_n$ must be 8-bit, 16-bit, or 32-bit general-purpose registers, as specified by the suffix.
`testb srcmem`$_8$`, destreg`$_8$ `testw srcmem`$_{16}$`, destreg`$_{16}$ `testl srcmem`$_{32}$`, destreg`$_{32}$	`destreg`$_n$ `- srcmem`$_n$ `(result to EFLAGS)` `destreg`$_n$ must be an 8-bit, 16-bit, or 32-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`$_{32}$	`destmem`$_n$ `- srcreg`$_n$ `(result to EFLAGS)` `srcreg`$_n$ must be an 8-bit, 16-bit, or 32-bit general-purpose register, as specified by the suffix; `destmem`$_n$ can be any like-sized memory location.

Table A-101: MASM/TASM Syntax for `test`

Instruction	Description
`test destreg, constant`	`destreg - constant (result to EFLAGS)` `destreg` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`test destmem, constant`	`destmem - constant (result to EFLAGS)` `destmem` must be an 8-bit, 16-bit, or 32-bit memory variable.
`test destreg, srcreg`	`destreg - srcreg (result to EFLAGS)` `destreg` and `srcreg` must be 8-bit, 16-bit, or 32-bit general-purpose registers. They must both be the same size.
`test destreg, srcmem`	`destreg - srcmem (result to EFLAGS)` `destreg` must be an 8-bit, 16-bit, or 32-bit general-purpose register. `srcmem` can be any like-sized memory location.
`test destmem, srcreg`	`destmem - srcreg (result to EFLAGS)` `srcreg` must be an 8-bit, 16-bit, or 32-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 one in the HO bit position.
Zero	Set if the logical AND of the two operands produces a zero result.

A.31 xor

The `xor` instruction requires two operands: a source operand and a destination operand. 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` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`xor( constant, destmem );`	`destmem := destmem XOR constant` `destmem` must be an 8-bit, 16-bit, or 32-bit memory variable.
`xor( srcreg, destreg );`	`destreg := destreg XOR srcreg` `destreg` and `srcreg` must be 8-bit, 16-bit, or 32-bit general-purpose registers. They must both be the same size.

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

Instruction	Description
`xor( srcmem, destreg );`	`destreg := destreg XOR srcmem` `destreg` must be an 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` must be an 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`$_n$ must be an 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`$_n$ must be an 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`$_{32}$	`destreg`$_n$ `:= destreg`$_n$ `XOR srcreg`$_n$ `destreg`$_n$ and `srcreg`$_n$ must be 8-bit, 16-bit, or 32-bit general-purpose registers, as specified by the suffix.
`xorb srcmem`$_8$`, destreg`$_8$ `xorw srcmem`$_{16}$`, destreg`$_{16}$ `xorl srcmem`$_{32}$`, destreg`$_{32}$	`destreg`$_n$ `:= destreg`$_n$ `XOR srcmem`$_n$ `destreg`$_n$ must be an 8-bit, 16-bit, or 32-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`$_{32}$	`destmem`$_n$ `:= destmem`$_n$ `XOR srcreg`$_n$ `srcreg`$_n$ must be an 8-bit, 16-bit, or 32-bit general-purpose register, as specified by the suffix; `destmem`$_n$ can be any like-sized memory location.

Table A-105: MASM/TASM Syntax for xor

Instruction	Description
`xor destreg, constant`	`destreg := destreg XOR constant` `destreg` must be an 8-bit, 16-bit, or 32-bit general-purpose register.
`xor destmem, constant`	`destmem := destmem XOR constant` `destmem` must be an 8-bit, 16-bit, or 32-bit memory variable.
`xor destreg, srcreg`	`destreg := destreg XOR srcreg` `destreg` and `srcreg` must be 8-bit, 16-bit, or 32-bit general-purpose registers. They must both be the same size.
`xor destreg, srcmem`	`destreg := destreg XOR srcmem` `destreg` must be an 8-bit, 16-bit, or 32-bit general-purpose register. `srcmem` can be any like-sized memory location.
`xor destmem, srcreg`	`destmem := destmem XOR srcreg` `srcreg` must be an 8-bit, 16-bit, or 32-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 one in the HO bit position.
Zero	Set if the logical XOR of the two operands produces a zero result.