Princeton University
COS 217: Introduction to Programming Systems
A Subset of IA-32 Assembly Language

1. Instruction Operands

1.1. Immediate Operands

Syntax: `$i`
Semantics: Evaluates to *i*. Note that *i* could be a label...

Syntax: `$label`
Semantics: Evaluates to the memory address denoted by *label*.

1.2. Register Operands

Syntax: `%r`
Semantics: Evaluates to reg[*r*], that is, the contents of register *r*.

1.3. Memory Operands

Syntax: `disp(%base, %index, scale)`

Semantics:
disp is a literal or label.
base is a general purpose register.
index is any general purpose register except EBP.
scale is the literal 1, 2, 4, or 8.

One of *disp*, *base*, or *index* is required. All other fields are optional.

Evaluates to the contents of memory at a certain address. The address is computed using this formula: *disp* + reg[*base*] + (reg[*index*] * *scale*)

The default *disp* is 0. The default *scale* is 1. If *base* is omitted, then reg[*base*] evaluates to 0. If *index* is omitted, then reg[*index*] evaluates to 0.

Syntax	Semantics	Description
`disp`	`mem[disp]`	**Direct Addressing**. Often used to access a long, word, or byte in the **bss**, **data**, or **rodata** section.
`(%base)`	`mem[reg[base]]`	**Indirect Addressing**. Often used to access a long, word, or byte in the **stack** section.
`disp(%base)`	`mem[disp+reg[base]]`	**Base+Displacement Addressing**. Often used to access a long, word, or byte in the **stack** section.
`disp(%base,%index)`	`mem[disp+reg[base]+reg[index]]`	**Indexed Addressing**. Often used to access an array of bytes (characters) in the **bss**, **data**, or **rodata** section.
`disp(%base,%index, scale)`	`mem[disp+reg[base]+(reg[index]*scale)]`	**Scaled Indexed Addressing**. Often used to access an array of longs or words in the **bss**, **data**, or **rodata** section.

2. Assembler Mnemonics

Key:

src: a source operand
dest: a destination operand
I: an immediate operand
R: a register operand
M: a memory operand
label: a label operand

For each instruction, at most one operand can be a memory operand.

2.1. Data Transfer Mnemonics

Syntax	Semantics	Description
`mov{l,w,b}` *`srcIRM, destRM`*	*`dest = src;`*	**Move**. Copy *src* to *dest.* Flags affected: None.
`movsb{l,w}` *`srcRM, destR`*	*`dest = src;`*	**Move Sign-Extended Byte**. Copy byte operand *src* to word or long operand *dest*, extending the sign of *src*. Flags affected:None.
`movswl` *`srcRM, destR`*	*`dest = src;`*	**Move Sign-Extended Word**. Copy word operand *src* to long operand *dest*, extending the sign of *src*. Flags affected:None.
`movzb{l,w}` *`srcRM, destR`*	*`dest = src;`*	**Move Zero-Extended Byte**. Copy byte operand *src* to word or long operand *dest*, setting the high-order bytes of *dest* to 0. Flags affected:None.
`movzwl` *`srcRM, destR`*	*`dest = src;`*	**Move Zero-Extended Word**. Copy word operand *src* to long operand *dest*, setting the high-order bytes of *dest* to 0. Flags affected: None.

`cmov{e,ne,`
 `l,le,g,ge,`
 `b,be,a,ae}`
 `srcRM, destR`
	`if (reg[EFLAGS} appropriate)`
	 `dest = src;`
		Conditional move. Copy long or word operand *src* to long or word register *dest* iff the flags in the EFLAGS register indicate a(n) equal to, unequal to, less than, less than or equal to, greater than, greater than, below, below or equal to, above, or above or equal to (respectively) relationship between the most recently compared numbers. The l, le, g, and ge forms are used after comparing signed numbers; the b, be, a, and ae forms are used after comparing unsigned numbers. Flags affected: None.

`push{l,w}` *`srcIRM`*	`reg[ESP] = reg[ESP] - {4,2};`
	 `mem[reg[ESP]] =` *`src;`*
		Push. Push *src* onto the stack. Flags affected: None.

`pop{l,w}` *`destRM`*	*`dest`* `= mem[reg[ESP]];`
	 `reg[ESP] = reg[ESP] + {4,2};`
		Pop. Pop from the stack into *dest.* Flags affected: None.

`lea{l,w}` *`srcM, destR`*	*`dest = &src;`*	**Load Effective Address**. Assign the address of *src* to *dest*. Flags affected: None.
`cltd`	`reg[EDX:EAX] = reg[EAX];`	**Convert Long to Double Register**. Sign extend the contents of register EAX into the register pair EDX:EAX, typically in preparation for idivl. Flags affected: None.
`cwtd`	`reg[DX:AX] = reg[AX];`	**Convert Word to Double Register.** Sign extend the contents of register AX into the register pair DX:AX, typically in preparation for idivw. Flags affected: None.

`cbtw`	`reg[AX] = reg[AL];`	**Convert Byte to Word.** Sign extend the contents of register AL into register AX, typically in preparation for idivb. Flags affected: None.

`leave`	`Equivalent to:`
	`movl %ebp, %esp`
	`popl %ebp`
		Leave. Pop a stack frame in preparation for leaving a function. Flags affected: None.

2.2. Arithmetic Mnemonics

Syntax	Semantics	Description
`add{l,w,b}` *srcIRM, destRM*	*dest = dest + src;*	**Add**. Add *src* to *dest.* Flags affected: O, S, Z, A, C, P.
`adc{l,w,b}` *srcIRM, destRM*	*dest = dest + src + C;*	**Add with Carry.** Add *src* and the C flag to *dest*. Flags affected: O, S, Z, A, C, P.
`sub{l,w,b}` *srcIRM, destRM*	*dest = dest - src;*	**Subtract**. Subtract *src* from *dest.* Flags affected: O, S, Z, A, C, P.
`inc{l,w,b}` *destRM*	*dest = dest + 1;*	**Increment**. Increment *dest.* Flags affected: O, S, Z, A, P.
`dec{l,w,b}` *destRM*	*dest = dest - 1;*	**Decrement**. Decrement *dest.* Flags affected: O, S, Z, A, P.
`neg{l,w,b}` *destRM*	*dest = -dest;*	**Negate**. Negate *dest.* Flags affected: O, S, Z, A, C, P.
`imull` *srcRM*	`reg[EDX:EAX] = reg[EAX]*`*src;*	**Signed Multiply**. Multiply the contents of register EAX by *src*, and store the product in registers EDX:EAX. Flags affected: O, S, Z, A, C, P.
`imulw` *srcRM*	`reg[DX:AX] = reg[AX]*`*src;*	**Signed Multiply**. Multiply the contents of register AX by *src*, and store the product in registers DX:AX. Flags affected: O, S, Z, A, C, P.
`imulb` *srcRM*	`reg[AX] = reg[AL]*`*src;*	**Signed Multiply**. Multiply the contents of register AL by *src*, and store the product in AX. Flags affected: O, S, Z, A, C, P.

`idivl` *srcRM*	`reg[EAX] = reg[EDX:EAX]/`*src;*
	`reg[EDX] = reg[EDX:EAX]%`*src;*
		Signed Divide. Divide the contents of registers EDX:EAX by *src*, and store the quotient in register EAX and the remainder in register EDX. Flags affected: O, S, Z, A, C, P.

`idivw` *srcRM*	`reg[AX] = reg[DX:AX]/`*src;*
	`reg[DX] = reg[DX:AX]%`*src;*
		Signed Divide. Divide the contents of registers DX:AX by *src*, and store the quotient in register AX and the remainder in register DX. Flags affected: O, S, Z, A, C, P.

`idivb` *srcRM*	`reg[AL] = reg[AX]/`*src;*
	`reg[AH] = reg[AX]%`*src;*
		Signed Divide. Divide the contents of register AX by *src*, and store the quotient in register AL and the remainder in register AH. Flags affected: O, S, Z, A, C, P.

`mull` *srcRM*	`reg[EDX:EAX] = reg[EAX]*`*src;*	**Unsigned Multiply**. Multiply the contents of register EAX by *src*, and store the product in registers EDX:EAX. Flags affected: O, S, Z, A, C, P.
`mulw` *srcRM*	`reg[DX:AX] = reg[AX]*`*src;*	**Unsigned Multiply**. Multiply the contents of register AX by *src*, and store the product in registers DX:AX. Flags affected: O, S, Z, A, C, P.
`mulb` *srcRM*	`reg[AX] = reg[AL]*`*src;*	**Unsigned Multiply**. Multiply the contents of register AL by *src*, and store the product in AX. Flags affected: O, S, Z, A, C, P.

`divl` *srcRM*	`reg[EAX] = reg[EDX:EAX]/`*src;*
	`reg[EDX] = reg[EDX:EAX]%`*src;*
		Unsigned Divide. Divide the contents of registers EDX:EAX by *src*, and store the quotient in register EAX and the remainder in register EDX. Flags affected: O, S, Z, A, C, P.

`divw srcRM`	`reg[AX] = reg[DX:AX]/src;`
	`reg[DX] = reg[DX:AX]%src;`
		Unsigned Divide. Divide the contents of registers DX:AX by *src*, and store the quotient in register AX and the remainder in register DX. Flags affected: O, S, Z, A, C, P.

`divb srcRM`	`reg[AL] = reg[AX]/src;`
	`reg[AH] = reg[AX]%src;`
		Unsigned Divide. Divide the contents of register AX by *src*, and store the quotient in register AL and the remainder in register AH. Flags affected: O, S, Z, A, C, P.

2.3. Bitwise Mnemonics

Syntax	Semantics	Description
`and{l,w,b} srcIRM, destRM`	`dest = dest & src;`	**And**. Bitwise and *src* into *dest.* Flags affected: O, S, Z, A, C, P.
`or{l,w,b} srcIRM, destRM`	`dest = dest | src;`	**Or**. Bitwise or *src* nito *dest.* Flags affected: O, S, Z, A, C, P.
`xor{l,w,b} srcIRM, destRM`	`dest = dest ^ src;`	**Exclusive Or**. Bitwise exclusive or *src* into *dest.* Flags affected: O, S, Z, A, C, P.
`not{l,w,b} destRM`	`dest = ~dest;`	**Not**. Bitwise not *dest.* Flags affected: None.
`sal{l,w,b} srcIR, destRM`	`dest = dest << src;`	**Shift Arithmetic Left**. Shift *dest* to the left *src* bits, filling with zeros. Flags affected: O, S, Z, A, C, P.
`sar{l,w,b} srcIR, destRM`	`dest = dest >> src;`	**Shift Arithmetic Right**. Shift *dest* to the right *src* bits, sign extending the number. Flags affected: O, S, Z, A, C, P.
`shl{l,w,b} srcIR, destRM`	`(Same as sal)`	**Shift Left**. (Same as sal.) Flags affected: O, S, Z, A, C, P.
`shr{l,w,b} srcIR, destRM`	`(Same as sar)`	**Shift Right**. Shift *dest* to the right *src* bits, filling with zeros. Flags affected: O, S, Z, A, C, P.

2.4. Control Transfer Mnemonics

Syntax	Semantics	Description

`cmp{l,w,b} srcIRM, destRM`	`reg[EFLAGS] =`
	`  dest comparedWith src;`
		Compare. Compute *dest* - *src* and set flags in the EFLAGS register based upon the result. Flags affected: O, S, Z, A, C, P.

`test{l,w,b} srcIRM, destRM`	`reg[EFLAGS] = dest & src;`	**Test**. Compute *dest* & *src* and set flags in the EFLAGS register based upon the result. Flags affected: S, Z, P (O and C set to 0).

`set{e,ne,`
`    l,le,g,ge,`
`    b,be,a,ae} destRM`
	`if (reg[EFLAGS] appropriate)`
	`   dest = 1;`
	`else`
	`   dest = 0;`
		Set. Set one-byte *dest* to 1 if the flags in the EFLAGS register indicate a(n) equal to, unequal to, less than, less than or equal to, greater than, greater than, below, below or equal to, above, or above or equal to (respectively) relationship between the most recently compared numbers. Otherwise set *destRM* to 0. The l, le, g, and ge forms are used after comparing signed numbers; the b, be, a, and ae forms are used after comparing unsigned numbers. Flags affected: None.

`jmp label`	`reg[EIP] = label;`	**Jump**. Jump to *label*. Flags affected: None.
`jmp *srcR`	`reg[EIP] = reg[src];`	**Jump indirect.** Jump to the address in *srcR*. Flags affected: None.

`j{e,ne,`
`  l,le,g,ge,`
`  b,be,a, ae} label`
	`if (reg[EFLAGS] appropriate)`
	`  reg[EIP] = label;`
		Conditional Jump. Jump to *label* iff the flags in the EFLAGS register indicate a(n) equal to, unequal to, less than, less than or equal to, greater than, greater than or equal to, below, below or equal to, above, or above or equal to (respectively) relationship between the most recently compared numbers. The l, le, g, and ge forms are used after comparing signed numbers; the b, be, a, and ae forms are used after comparing unsigned numbers. Flags affected: None.

`call label`	`reg[ESP] = reg[ESP] - 4;`
	`mem[reg[ESP]] = reg[EIP];`
	`reg[EIP] = label;`
		Call. Call the function that begins at *label.* Flags affected: None.

`call *srcR`	`reg[ESP] = reg[ESP] - 4;`
	`mem[reg[ESP]] = reg[EIP];`
	`reg[EIP] = reg[src];`
		Call indirect. Call the function whose address is in *src*. Flags affected: None.

`ret`	`reg[EIP] = mem[reg[ESP]];`
	`reg[ESP] = reg[ESP] + 4;`
		Return. Return from the current function. Flags affected: None.

`int srcIRM`	`Generate interrupt number src`	**Interrupt**. Generate interrupt number *src*. Flags affected: None.

3. Assembler Directives

Syntax	Description
`label:`	Record the fact that *label* marks the current location within the current section.
`.section ".sectionname"`	Make the *sectionname* section the current section.
`.skip n`	Skip *n* bytes of memory in the current section.
`.align n`	Skip as many bytes of memory in the current section as necessary so the current location is evenly divisible by *n*.
`.byte bytevalue1, bytevalue2, ...`	Allocate one byte of memory containing *bytevalue1*, one byte of memory containing *bytevalue2*, ... in the current section.
`.word wordvalue1, wordvalue2, ...`	Allocate two bytes of memory containing *wordvalue1*, two bytes of memory containing *wordvalue2*, ... in the current section.
`.long longvalue1, longvalue2, ...`	Allocate four bytes of memory containing *longvalue1*, four bytes of memory containing *longvalue2*, ... in the current section.
`.ascii "string1", "string2", ...`	Allocate memory containing the characters from *string1*, *string2*, ... in the current section.
`.asciz "string1", "string2", ...`	Allocate memory containing *string1*, *string2*, ..., where each string is '\0' terminated, in the current section.
`.string "string1", "string2", ...`	Same as .asciz.
`.globl label1, label2, ...`	Mark *label1*, *label2*, ... so they are accessible by code generated from other source code files.
`.equ name, expr`	Define *name* as a symbolic alias for *expr*.
`.lcomm label, n [,align]`	Allocate *n* bytes, marked by *label*, in the bss section [and align the bytes on an *align*-byte boundary].
`.comm label, n, [,align]`	Allocate *n* bytes, marked by *label*, in the bss section, mark label so it is accessible by code generated from other source code files [and align the bytes on an *align*-byte boundary].
`.type label,@function`	Mark *label* so the linker knows that it denotes the beginning of a function.