

Princeton University

COS 217: Introduction to Programming Systems

A Subset of IA-32 Assembly Language

Instruction Operands

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 $label$.

Register Operands

Syntax: $\%r$

Semantics: Evaluates to $\text{reg}[r]$, that is, the contents of register r .

Memory Operands

Syntax: $\%section:disp(\%base, \%index, scale)$

Semantics:

$section$ is a section register (CS, SS, DS, or ES).

$disp$ is a literal or label.

$base$ is a general-purpose register.

$index$ is any general purpose register except EBP.

$scale$ is the literal 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. That address consists of an offset into a section.

The section is specified by $section$. Assembly language programmers typically rely on the default section:

- CS for instruction fetches.
- SS for stack pushes and pops and references using ESP or EBP as base.
- DS for all data references except when relative to a stack or string destination.
- ES for the destinations of all string instructions.

The offset is computed using this expression:

$$\text{reg}[base] + (\text{reg}[index] * scale) + disp$$

The default $disp$ is 0. The default $scale$ is 0. If $base$ is omitted, then $\text{reg}[base]$ evaluates to 0. If $index$ is omitted, then $\text{reg}[index]$ evaluates to 0.

Commonly Used Memory Operands

Syntax	Semantics	Description
<i>label</i>	disp: <i>label</i> base: (none) index: (none) scale: (none) mem[0+(0*0)+ <i>label</i>] mem[<i>label</i>]	<p>Direct Addressing. The contents of memory at a certain address. The offset of that address is denoted by <i>label</i>.</p> <p>Often used to access a long, word, or byte in the bss, data, or rodata section.</p>
(% <i>r</i>)	disp: (none) base: <i>r</i> index: (none) scale: (none) mem[reg[<i>r</i>]+(0*0)+0] mem[reg[<i>r</i>]]	<p>Indirect Addressing. The contents of memory at a certain address. The offset of that address is the contents of register <i>r</i>.</p> <p>Often used to access a long, word, or byte in the stack section.</p>
<i>i</i> (% <i>r</i>)	disp: <i>i</i> base: <i>r</i> index: (none) scale: (none) mem[reg[<i>r</i>]+(0*0)+ <i>i</i>] mem[reg[<i>r</i>]+ <i>i</i>]	<p>Base-Pointer Addressing. The contents of memory at a certain address. The offset of that address is the sum of <i>i</i> and the contents of register <i>r</i>.</p> <p>Often used to access a long, word, or byte in the stack section.</p>
<i>label</i> (% <i>r</i>)	disp: <i>label</i> base: <i>r</i> index: (none) scale: (none) mem[reg[<i>r</i>]+(0*0)+ <i>label</i>] mem[reg[<i>r</i>]+ <i>label</i>]	<p>Indexed Addressing. The contents of memory at a certain address. The offset of that address is the sum of the address denoted by <i>label</i> and the contents of register <i>r</i>.</p> <p>Often used to access an array of bytes (characters) in the bss, data, or rodata section.</p>
<i>label</i> (,% <i>r</i> , <i>i</i>)	disp: <i>label</i> base: (none) index: <i>r</i> scale: <i>i</i> mem[0+(reg[<i>r</i>]* <i>i</i>)+ <i>label</i>] mem[(reg[<i>r</i>]* <i>i</i>)+ <i>label</i>]	<p>Indexed Addressing. The contents of memory at a certain address. The offset of that address is the sum of the address denoted by <i>label</i>, and the contents of register <i>r</i> multiplied by <i>i</i>.</p> <p>Often used to access an array of longs or words in the bss, data, or rodata section.</p>

Instructions

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

Syntax	Semantics (expressed using C-like syntax)	Description
Data Transfer		
<code>mov{1,w,b} srcIRM, destRM</code>	<code>dest = src;</code>	Move. Copy <i>src</i> to <i>dest</i> .
<code>push{1,w} srcIRM</code>	<code>reg[ESP] = reg[ESP] - {4,2}; mem[reg[ESP]] = src;</code>	Push. Push <i>src</i> onto the stack.
<code>pop{1,w} destRM</code>	<code>dest = mem[reg[ESP]]; reg[ESP] = reg[ESP] + {4,2};</code>	Pop. Pop from the stack into <i>dest</i> .
<code>lea{1,w} srcM, destR</code>	<code>dest = &src;</code>	Load Effective Address. Assign the address of <i>src</i> to <i>dest</i> .
<code>cld</code>	<code>reg[EDX:EAX] = reg[EAX];</code>	Convert Long to Double Register. Sign extend the contents of register EAX into the register pair EDX:EAX, typically in preparation for <code>idivl</code> .
<code>cwtd</code>	<code>reg[DX:AX] = reg[AX];</code>	Convert Word to Double Register. Sign extend the contents of register AX into the register pair DX:AX, typically in preparation for <code>idivw</code> .
<code>cbtw</code>	<code>reg[AX] = reg[AL];</code>	Convert Byte to Word. Sign extend the contents of register AL into register AX, typically in preparation for <code>idivb</code> .
<code>leave</code>	Equivalent to: <code>movl %ebp, %esp popl %ebp</code>	Pop a stack frame in preparation for leaving a function
Arithmetic		
<code>add{1,w,b} srcIRM, destRM</code>	<code>dest = dest + src;</code>	Add. Add <i>src</i> to <i>dest</i> .
<code>sub{1,w,b} srcIRM, destRM</code>	<code>dest = dest - src;</code>	Subtract. Subtract <i>src</i> from <i>dest</i> .
<code>inc{1,w,b} destRM</code>	<code>dest = dest + 1;</code>	Increment. Increment <i>dest</i> .
<code>dec{1,w,b} destRM</code>	<code>dest = dest - 1;</code>	Decrement. Decrement <i>dest</i> .
<code>neg{1,w,b} destRM</code>	<code>dest = -dest;</code>	Negate. Negate <i>dest</i> .
<code>imull srcRM</code>	<code>reg[EDX:EAX] = reg[EAX]*src;</code>	Multiply. Multiply the contents of register EAX by <i>src</i> , and store the product in registers EDX:EAX.
<code>imulw srcRM</code>	<code>reg[DX:AX] = reg[AX]*src;</code>	Multiply. Multiply the contents of register AX by <i>src</i> , and store the product in registers DX:AX.
<code>imulb srcRM</code>	<code>reg[AX] = reg[AL]*src;</code>	Multiply. Multiply the contents of register AL by <i>src</i> , and store the product in AX.
<code>idivl srcRM</code>	<code>reg[EAX] = reg[EDX:EAX]/src; reg[EDX] = reg[EDX:EAX]%src;</code>	Divide. Divide the contents of registers EDX:EAX by <i>src</i> , and store the quotient in register EAX and the remainder in register EDX.

<code>idivw srcRM</code>	<code>reg[AX] = reg[DX:AX]/src;</code> <code>reg[DX] = reg[DX:AX]%src;</code>	Divide. Divide the contents of registers DX:AX by <i>src</i> , and store the quotient in register AX and the remainder in register DX.
<code>idivb srcRM</code>	<code>reg[AL] = reg[AX]/src;</code> <code>reg[AH] = reg[AX]%src;</code>	Divide. Divide the contents of register AX by <i>src</i> , and store the quotient in register AL and the remainder in register AH.
Bitwise		
<code>and{l,w,b} srcIRM, destRM</code>	<code>dest = dest & src;</code>	And. Bitwise and <i>src</i> into <i>dest</i> .
<code>or{l,w,b} srcIRM, destRM</code>	<code>dest = dest src;</code>	Or. Bitwise or <i>src</i> into <i>dest</i> .
<code>xor{l,w,b} srcIRM, destRM</code>	<code>dest = dest ^ src;</code>	Exclusive Or. Bitwise exclusive or <i>src</i> into <i>dest</i> .
<code>not{l,w,b} destRM</code>	<code>dest = ~dest;</code>	Not. Bitwise not <i>dest</i> .
<code>sal{l,w,b} srcIR, destRM</code>	<code>dest = dest << src;</code>	Shift Arithmetic Left. Shift <i>dest</i> to the left <i>src</i> bits, filling with zeros.
<code>sar{l,w,b} srcIR, destRM</code>	<code>dest = dest >> src;</code>	Shift Arithmetic Right. Shift <i>dest</i> to the right <i>src</i> bits, sign extending the number.
<code>shl{l,w,b} srcIR, destRM</code>	(Same as <code>sal</code>)	Shift Left. (Same as <code>sal</code> .)
<code>shr{l,w,b} srcIR, destRM</code>	(Same as <code>sar</code>)	Shift Right. Shift <i>dest</i> to the right <i>src</i> bits, filling with zeros.
Control Transfer		
<code>cmp{l,w,b} srcIRM1,srcRM2</code>	<code>reg[EFLAGS] =</code> <code>srcRM2 comparedwith</code> <code>srcIRM1</code>	Compare. Compare <i>src2</i> with <i>src1</i> , and set the condition codes in the EFLAGS register accordingly.
<code>jmp label</code>	<code>reg[EIP] = label;</code>	Jump. Jump to label.
<code>j{e,ne,l,le,g,ge} label</code>	<code>if (reg[EFLAGS] appropriate)</code> <code>reg[EIP] = label;</code>	Conditional Jump. Jump to label iff the condition codes in the EFLAGS register are set appropriately.
<code>call label</code>	<code>reg[ESP] = reg[ESP] - 4;</code> <code>mem[reg[ESP]] = reg[EIP];</code> <code>reg[EIP] = label;</code>	Call. Call the function that begins at <i>label</i> .
<code>call *srcR</code>	<code>reg[ESP] = reg[ESP] - 4;</code> <code>mem[reg[ESP]] = reg[EIP];</code> <code>reg[EIP] = reg[srcR];</code>	Call. Call the function whose address is in <i>src</i> .
<code>ret</code>	<code>reg[EIP] = mem[reg[ESP]];</code> <code>reg[ESP] = reg[ESP] + 4;</code>	Return. Return from the current function.
<code>int srcIRM</code>	Generate interrupt number <i>src</i>	Interrupt. Generate interrupt number <i>src</i> .

Assembler Directives

Syntax	Description
<code>label:</code>	Record the fact that <i>label</i> marks the current location within the current section
<code>.section ".sectionname"</code>	Make the <i>sectionname</i> section the current section
<code>.skip n</code>	Skip <i>n</i> bytes of memory in the current section
<code>.byte bytevalue1, bytevalue2, ...</code>	Allocate one byte of memory containing <i>bytevalue1</i> , one byte of memory containing <i>bytevalue2</i> , ... in the current section
<code>.word wordvalue1, wordvalue2, ...</code>	Allocate two bytes of memory containing <i>wordvalue1</i> , two bytes of memory containing <i>wordvalue2</i> , ... in the current section
<code>.long longvalue1, longvalue2, ...</code>	Allocate four bytes of memory containing <i>longvalue1</i> , four bytes of memory containing <i>longvalue2</i> , ... in the current section
<code>.ascii "string1", "string2", ...</code>	Allocate memory containing the characters from <i>string1</i> , <i>string2</i> , ... in the current section
<code>.asciz "string1", "string2", ...</code>	Allocate memory containing <i>string1</i> , <i>string2</i> , ..., where each string is NULL terminated, in the current section
<code>.string "string1", "string2", ...</code>	(Same as <code>.asciz</code>)
<code>.globl label1, label2, ...</code>	Mark <i>label1</i> , <i>label2</i> , ... so they are available to the linker
<code>.equ name, value</code>	Define <i>name</i> as a symbolic alias for <i>value</i>
<code>.type label,@function</code>	Mark <i>label</i> so the linker knows that it denotes the beginning of a function

Copyright © 2004 by Robert M. Dondero, Jr.