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 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:

reg[*base*] + (reg[*index*] * *scale*) + *disp*

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

Commonly Used Memory Operands

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

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.

Syntax	Semantics (expressed using C-like syntax)	Description
Data Transfer		
`mov{l,w,b} srcIRM, destRM`	`dest = src;`	**Move**. Copy *src* to *dest.*
`push{l,w} srcIRM`	`reg[ESP] = reg[ESP] - {4,2};` `mem[reg[ESP]] = src;`	**Push**. Push *src* onto the stack.
`pop{l,w} destRM`	`dest = mem[reg[ESP]];` `reg[ESP] = reg[ESP] + {4,2};`	**Pop**. Pop from the stack into *dest.*
`lea{l,w} srcM, destR`	`dest = &src;`	**Load Effective Address**. Assign the address of *src* to *dest*.
`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.
`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.
`cbtw`	`reg[AX] = reg[AL];`	**Convert Byte to Word.** Sign extend the contents of register AL into register AX, typically in preparation for idivb.

`leave`	`Equivalent to:`
	`  movl  %ebp, %esp`
	`  popl  %ebp`
		Pop a stack frame in preparation for **leaving** a function

Arithmetic		
`add{l,w,b} srcIRM, destRM`	`dest = dest + src;`	**Add**. Add *src* to *dest.*
`sub{l,w,b} srcIRM, destRM`	`dest = dest - src;`	**Subtract**. Subtract *src* from *dest.*
`inc{l,w,b} destRM`	`dest = dest + 1;`	**Increment**. Increment *dest.*
`dec{l,w,b} destRM`	`dest = dest - 1;`	**Decrement**. Decrement *dest.*
`neg{l,w,b} destRM`	`dest = -dest;`	**Negate**. Negate *dest.*
`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.
`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.
`imulb srcRM`	`reg[AX] = reg[AL]*src;`	**Signed Multiply**. Multiply the contents of register AL by *src*, and store the product in AX.
`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.

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.
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.
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.
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.
mulb *srcRM*	reg[AX] = reg[AL]**src*;	**Unsigned Multiply**. Multiply the contents of register AL by *src*, and store the product in AX.
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.
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.
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.
Bitwise		
and{l,w,b} *srcIRM, destRM*	*dest = dest & src;*	**And**. Bitwise and *src* into *dest.*
or{l,w,b} *srcIRM, destRM*	*dest = dest | src;*	**Or**. Bitwise or *src* nito *dest.*
xor{l,w,b} *srcIRM, destRM*	*dest = dest ^ src;*	**Exclusive Or**. Bitwise exclusive or *src* into *dest.*
not{l,w,b} *destRM*	*dest = ~dest;*	**Not**. Bitwise not *dest.*
sal{l,w,b} *srcIR, destRM*	*dest = dest << src;*	**Shift Arithmetic Left**. Shift *dest* to the left *src* bits, filling with zeros.
sar{l,w,b} *srcIR, destRM*	*dest = dest >> src;*	**Shift Arithmetic Right**. Shift *dest* to the right *src* bits, sign extending the number.
shl{l,w,b} *srcIR, destRM*	(Same as sal)	**Shift Left**. (Same as sal.)
shr{l,w,b} *srcIR, destRM*	(Same as sar)	**Shift Right**. Shift *dest* to the right *src* bits, filling with zeros.
Control Transfer		
cmp{l,w,b} *srcIRM1,srcRM2*	reg[EFLAGS] = *srcRM2* comparedwith *srcIRM1*	**Compare**. Compare *src2* with *src1*, and set the condition codes in the EFLAGS register accordingly.
jmp *label*	reg[EIP] = *label*;	**Jump**. Jump to *label*.
j{e,ne} *label*	if (reg[EFLAGS] appropriate) reg[EIP] = *label*;	**Conditional Jump**. Jump to *label* iff the condition codes in the EFLAGS register indicate an equality or inequality (respectively) relationship between the most recently compared numbers.
j{l,le,g,ge} *label*	if (reg[EFLAGS] appropriate) reg[EIP] = *label*;	**Signed Conditional Jump**. Jump to *label* iff the condition codes in the EFLAGS register indicate a less than, less than or equal to, greater than, or greater than or equal to (respectively) relationship between the most recently compared numbers.
j{b,be,a,ae} *label*	if (reg[EFLAGS] appropriate) reg[EIP] = *label*;	**Unsigned Conditional Jump**. Jump to *label* iff the condition codes in the EFLAGS register indicate a below, below

		or equal to, above, or above or equal to (respectively) relationship between the most recently compared numbers.

`call label`	`reg[ESP] = reg[ESP] - 4;`
	`mem[reg[ESP]] = reg[EIP];`
	`reg[EIP] = label;`
		Call. Call the function that begins at *label*.

`call *srcR`	`reg[ESP] = reg[ESP] - 4;`
	`mem[reg[ESP]] = reg[EIP];`
	`reg[EIP] = reg[srcR];`
		Call. Call the function whose address is in *src*.

`ret`	`reg[EIP] = mem[reg[ESP]];` `reg[ESP] = reg[ESP] + 4;`	**Return**. Return from the current function.
`int srcIRM`	`Generate interrupt number src`	**Interrupt**. Generate interrupt number *src*.

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 NULL terminated, in the current section
`.string "string1", "string2", ...`	(Same as .asciz)
`.globl label1, label2, ...`	Mark *label1*, *label2*, ... so they are available to the linker
`.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 available to the linker [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