

# *CALCOLATORI*

## *Cenni ad Assembly Intel*

Giovanni Iacca  
[giovanni.iacca@unitn.it](mailto:giovanni.iacca@unitn.it)

*Lezione basata su materiale preparato  
da Luca Abeni, Luigi Palopoli, Fabiano Zenatti e Marco Roveri*



UNIVERSITÀ DEGLI STUDI DI TRENTO

---

Dipartimento di Ingegneria  
e Scienza dell'Informazione

# Architettura Intel

- Utilizzata sulla maggior parte dei laptop, desktop e server moderni
- Lunga storia
  - Dagli anni '70 (Intel 8080 - 8 bit!)...
  - ...Fino ad oggi (Intel Core i7, i9 e simili, 64 bit!)
- E la storia ha dei retaggi
  - Evoluzione lenta ma costante, “guardando al passato”...
  - Moderne CPU a 64 bit sono ancora in grado di eseguire vecchio codice ad 8 bit (MSDOS, anyone?)!
    - Immaginate complicazioni e complessità che questo comporta...
- Architettura CISC (Complex Instruction Set Computer), contrapposta a RISC (Reduced Instruction Set Computer) di RISC-V (e MIPS)

# Assembly Intel

- CISC → gran numero di istruzioni e modalità di indirizzamento
  - Non vedremo tutto in modo esaustivo (ottimi tutorial online)
  - Ci focalizzeremo su differenze rispetto a RISC-V
- Compatibilità → varie modalità di funzionamento
  - Considereremo solo modalità a 64 bit e relative ISA e ABI
- Esistono vari Assembler, ognuno con sintassi differenti
  - Considereremo GNU Assembler (`gas`)

# Registri General Purpose

- Sono tutti preceduti dal prefisso %
- 16 registri a 64 bit (più o meno) general purpose
- Hanno nomi che riflettono gli strascichi di compatibilità con le versioni precedenti:
  - `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rsi`, `%rdi`, `%rbp`, `%rsp`
  - `%r8`, ..., `%r15`
- Note:
  - `%rsp` → stack pointer;
  - `%rbp` → base pointer (puntatore a stack frame)
  - `%rsi` e `%rdi` derivano da due registri `%si` e `%di` della CPU 8086 per la copia di array (`%si` → source index, `%di` → destination index)

# Registri General Purpose - 2

I registri `%rax ... %rsp` estendono registri a 32 bit (`%eax ... %esp`)

- ...Che a loro volta estendono rispettivi registri a 16 bit (`%ax ... %sp`)
- ...Ognuno dei primi quattro registri (`%ax, %bx, %cx, e %dx`) sono composti da 2 registri a 8 bit indicati sostituendo `%x` con `%h` o `%l`):
  - `%ax == %ah + %al`
- ... Ognuno dei registri `r8, ... , r15` estende il rispettivo registro a 32 bit (`r8d, ... , r15d`), che a loro volta estende rispettivo registro a 16 bit (`r8w, ... , r15w`) che a loro volta contiene un registro a 8 bit (`r8b, ... , r15b`).
  - In questo caso c'è solo un registro a 8 bit (byte meno significativo)

63	31	15	8	7	0
%rax	%eax	%ax	%ah	%al	
%rbx	%ebx	%bx	%bh	%bl	
%rcx	%ecx	%cx	%ch	%cl	
%rdx	%edx	%dx	%dh	%dl	
%rsi	%esi	%si		%sil	
%rdi	%edi	%di		%dil	
%rbp	%ebp	%bp		%bpl	
%rsp	%esp	%sp		%spl	
%r8	%r8d	%r8w		%r8b	
%r9	%r9d	%r9w		%r9b	
...					
%r15	%r15d	%r15w		%r15b	

# Registri “Speciali”

- Instruction Pointer “visibile”: `%rip`
- Flags register: `%rflags` (estende `%eflags`, che estende `%flags`)
- Set di flag settati da istruzioni logico aritmetiche
  - `CF` : Carry Flag → Messo a 1 se risultato è andato in unsigned overflow o se c'è carry-out
  - `ZF` : Zero Flag → Messo a 1 se risultato è zero
  - `SF` : Sign Flag → Messo a 1 se risultato è negativo
  - `OF` : 2's Overflow Flag → Messo a 1 se risultato è andato in overflow
- Usati da istruzioni di salto condizionale
- Altri flag controllano il funzionamento della CPU (`IF`) o contengono altre informazioni sulla CPU
- Equivalente a Program Status Word (PSW) di altre CPU
- Altri registri “speciali” non sono interessanti per noi ora

# Convenzioni di Chiamata

- Non fanno propriamente parte dell'architettura
  - Data una CPU / architettura, si possono usare molte diverse convenzioni di chiamata
  - Servono per “mettere d'accordo” diversi compilatori / librerie ed altre parti del Sistema Operativo
- Tecnicamente, sono specificate dall'ABI, non dall'ISA!!!
- Come / dove passare i parametri
  - Stack o registri?
- Quali registri preservare?
  - Quando un programma invoca una subroutine, quali registri può aspettarsi che contengano sempre lo stesso valore al ritorno?

# Convenzioni di Chiamata

- Primi 6 argomenti:
  - `%rdi, %rsi, %rdx, %rcx, %r8 ed %r9`
- Altri argomenti ( $7 \rightarrow n$ ): sullo stack
- Valori di ritorno:
  - `%rax e %rdx`
- Registri preservati:
  - `%rbp, %rbx, %r12, %r13, %r14 ed %r15`
- Registri non preservati:
  - `%rax, %r10, %r11, oltre ai registri per passaggio parametri: %rdi, %rsi, %rdx, %rcx, %r8 ed %r9`



# Modalità di Indirizzamento - 1

- Istruzioni prevalentemente a 2 operandi
  - Secondo operando: destinazione implicita!
  - Limitazione rispetto a RISC-V: impossibile specificare due operandi ed una destinazione diversa
- Sorgente (primo operando)
  - Operando Immediato (una costante con `$` come prefisso: e.g. `$20`)
  - Operando in Registro (valore di un registro e.g. `%rax`)
  - Operando in Memoria (valore in locazione di memoria e.g. valore ad indirizzo `0x0100A8`)
- Destinazione (secondo operando)
  - Operando in Registro (un registro come destinazione e.g. `%rdx`)
  - Operando in Memoria (una locazione di memoria specificata dall'indirizzo, locazione ad indirizzo `0x0AA0E2`)
- Possibili operazioni che consentono di scrivere sia su registri che memoria!
  - RISC-V: accesso a memoria solo per `load` e `store`
  - Vincolo: no entrambi gli operandi in memoria

## Modalità di Indirizzamento - 2

- Operandi in memoria: varie modalità di indirizzamento
- Semplificando un po': **accesso indiretto**
- Accesso a locazione di memoria
- Sintassi: `<displ> (<base reg>, <index reg>, <scale>)`
  - `<displacement>`: costante (valore immediato) a 8, 16 o 32 bit
    - simile a RISC-V - ma **RISC-V ha displacement/offset solo a 12 bit**
  - `<base>`: valore in registro (come per RISC-V)
  - `<indice>`: valore in registro (semplifica iterazione su array)
  - `<scala>`: valore costante: 1, 2, 4, o 8 (semplifica accesso ad array con elementi di dimensione 1, 2, 4 o 8 byte)
- **Indirizzo** `<displacement> + <base> + <indice> * <scala>`

## Indirizzamento - Casi Speciali

- Scala = 1 (no scale): `<displ> (<base reg>, <index reg>)`
- Niente scala e indice: `<displ> (<base reg>)`
  - Ricorda accesso per RISC-V. Unica differenza: la dimensione in bit dell'offset/displacement
- Niente scala, indice e displacement (displacement = 0): `(<base reg>)`
- Niente displacement (displacement = 0): `(<base reg>, <index reg>, <scale>)`
- ...

## Indirizzamento - limitazioni

- Entrambi gli operandi non possono essere in memoria
  - **movl** 345, (%eax) non è consentito perchè entrambe le destinazioni sono in memoria:  $\text{Mem}[\text{eax}] = \text{Mem}[345]$
  - La scrittura da memoria su memoria viene simulata con due istruzioni che usano un registro di appoggio:  
**movl** 345, %eax  
**movl** %eax, (%ebx)
- Combinazioni valide sono:
  - Imm → Reg
  - Imm → Mem
  - Reg → Reg
  - Mem → Reg
  - Reg → Mem

## Indirizzamento - riassunto

Nome	Forma	Esempio	Descrizione
Immediata	\$Num	<b>movl</b> \$-500, %rax	rax = \$Num
Accesso diretto	Num	<b>movl</b> 500, %rax	rax = Mem[Num]
Registro	$r_i$	<b>movl</b> %rdx, %rax	rax = rdx
Accesso indiretto	( $r_i$ )	<b>movl</b> (%rdx), %rax	rax = Mem[rdx]
Base e spiazzamento	Num( $r_i$ )	<b>movl</b> 31(%rdx), %rax	rax = Mem[rdx+31]
Indice scalato	( $r_b, r_i, s$ )	<b>movl</b> (%rdx, %rcx, 4), %rax	rax = Mem[rdx+rcx*4]
Indice scalato + spiazzamento	Num( $r_b, r_i, s$ )	<b>movl</b> 35(%rdx, %rcx, 4), %rax	rax = Mem[rdx+rcx*4+35]

$s$  è il fattore di scala e può assumere i valori: 1, 2, 4, o 8

# Istruzioni Intel

- Troppe per vederle tutte
  - Esistono diversi e buoni tutorial online
- Per una lista più o meno dettagliata delle istruzioni  
[http://en.wikipedia.org/wiki/X86\\_instruction\\_listings](http://en.wikipedia.org/wiki/X86_instruction_listings)
  - Molte istruzioni “non proprio utili” sono state mantenute per compatibilità!
- In genere, sintassi `<opcode> <source>, <destination>`
  - Carattere finale di opcode (`b` : 8bit, `w` : 16bit, `l` : 32bit, `q` : 64bit) per indicare “larghezza” (in bit) degli operandi
  - Nomi registri iniziano con “%”
  - Valori immediati (costanti) iniziano con “\$” (e.g. `$231`)
  - Indirizzi diretti (costanti) sono semplici numeri (e.g. “`439`”)

# Istruzioni Più Comuni - 1

- `mov`: copia dati da sorgente a destinazione
  - `movsx`: copia dati con estensione del segno
  - `movzx`: copia dati con estensione con 0
- `add / adc` (add with carry)
- `sub / sbc` (sub with carry)
- `mul / imul`: signed / unsigned multiplication
- `div / idiv`: signed / unsigned division
- `inc / dec`: somma / sottrae 1
- `rcl, rcr, rol, ror`: varie forme di “rotate”
- `sal, sar, shl, shr`: shift (aritmetico e logico)
- `and / or / xor / not`: operazioni booleane bit a bit
  - Istruzioni aritmetico / logiche modificano flag (carry, zero, sign, ...)
  - Altre istruzioni per modificare flag: `clic / stc, cld, cmc...`
- `neg`: complemento a 2 (negazione)
- `nop`

## Istruzioni Più Comuni - 3

- `push`: inserisce dati sullo stack
  - Sintassi: **`pushq %reg`**
    - Decrementa `%rsp` di 8
    - Scrive `%reg` nella memoria specificata in `%rsp`
  - Esempio: **`pushq %rax`**
    - Equivale a: **`subq $8, %rsp`**  
**`movq %rax, (%rsp)`**
- `pop`: rimuove dati da stack
  - Sintassi: **`popq %reg`**
    - Legge memoria all'indirizzo specificata in `%rsp` e memorizza valore in `%reg`
    - Incrementa `%rsp` di 8
  - Esempio: **`popq %rdx`**
    - Equivale a: **`movq (%rsp), %rdx`**  
**`addq $8, %rsp`**



## Istruzioni Più Comuni - 2

- `cmp` e `test`
  - `cmp`: **cmp** arg1, arg2
    - Compara arg2 con arg1 (e.g.  $\text{arg2} < \text{arg1}$ ,  $\text{arg2} = \text{arg1}$ , ...)
    - Effettua  $\text{arg2} - \text{arg1}$  e setta flags del flag register
    - arg1 e arg2 non sono modificati (risultato di sottrazione non memorizzato)
  - `test`: **test** arg1, arg2
    - Compara arg2 con arg1 (e.g.  $\text{arg2} = \text{arg1}$ , ...)
    - Effettua  $\text{arg2} \& \text{arg1}$  e setta flags del flag register
    - arg1 e arg2 non sono modificati (risultato di and bit a bit non memorizzato)
    - Spesso usato con  $\text{arg1} = \text{arg2}$  (e.g. **test** %eax, %eax per verificare se il valore è 0 o negativo (ZF o SF)).
- Da usarsi prima di salti condizionali

## Istruzioni Più Comuni - 3

- `jmp` : salto incondizionato
- `je` / `jnz` / `jc` / `jnc`: salti condizionati
  - La condizione di salto è stabilita in base ai valori nel flag register (jump if equal, jump if not zero, jump if carry, ...)
  - In generale, “`j<condition>`”

Istruzione	Sinonimo	Cond. Flags	Descrizione
<b>je</b> label	<b>jz</b> label	ZF	Uguali o zero
<b>jne</b> label	<b>jnz</b> label	$\sim ZF$	Differenti o Non zero
<b>js</b> label		SF	Negativo
<b>jns</b> label		$\sim SF$	Non Negativo
<b>jg</b> label	<b>jnle</b> label	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Signed >
<b>jge</b> label	<b>jnl</b> label	$\sim (SF \wedge OF)$	Signed >=
<b>jl</b> label	<b>jnge</b> label	$(SF \wedge OF)$	Signed <
<b>jle</b> label	<b>jng</b> label	$(SF \wedge OF) \mid ZF$	Signed <=
<b>ja</b> label	<b>jnbe</b> label	$\sim CF \ \& \ \sim ZF$	Unsigned >
<b>jae</b> label	<b>jnb</b> label	$\sim CF$	Unsigned >=
<b>jb</b> label	<b>jnae</b> label	CF	Unsigned <
<b>jbe</b> label	<b>jna</b> label	CF $\mid$ ZF	Unsigned <=

- Varie istruzioni “condizionali” (conditional move, set on condition, ...)

## Istruzioni Più Comuni - 4

- `call` chiamata a procedura
  - Sintassi: **call** label
    - Fa push sullo stack dell'indirizzo della prossima istruzione
    - Modifica il program counter per andare all'inizio della procedura desiderata (specificato con una label)
    - Implicitamente esegue: **subq** \$8, %rsp  
**movq** %rip, (%rsp)
- `ret`: ritorno da procedura
  - Sintassi: **ret**
    - Fa pop dallo stack dell'indirizzo di ritorno e lo memorizza in `%rip`
    - Modifica il program counter per andare alla prossima istruzione del chiamante
    - Implicitamente esegue: **movq** (%rsp), %rip  
**addq** \$8, %rsp

## Istruzioni Più Comuni - Esempi `mov`

- Sintassi: **`mov[b,w,l,q]`** src, dst
- Condizioni iniziali:

Mem[0x00204] = 7654 3210

Mem[0x00200] = fedc ba98

rax = ffff ffff 1234 5678

- **`movl`** 0x204, %eax
- **`movw`** 0x202, %ax
- **`movb`** 0x207, %al
- **`movq`** 0x200, %rax
- **`movb`** %al, 0x4e5
  
- **`movl`** %eax, 0x4e0

rax = 0000 0000 7654 3210

rax = 0000 0000 7654 fedc

rax = 0000 0000 7654 fe76

rax = 7654 3210 fedc ba98

Mem[0x004e4] = 0000 9800

Mem[0x004e0] = 0000 0000

Mem[0x004e4] = 0000 9800

Mem[0x004e0] = fedc ba98

## Istruzioni Più Comuni - Esempi `mov` - 2

- Sintassi: **`mov[b,w,l,q]`** src, dst
- Condizioni iniziali:

Mem[0x00204] = 7654 3210

Mem[0x00200] = fedc ba98

rax = ffff ffff 1234 5678

- **`movl $0xfe1234, %eax`**

rax = 0000 0000 00fe 1234

- **`movw $0xaa55, %ax`**

rax = 0000 0000 00fe aa55

- **`movb $20, %al`**

rax = 0000 0000 00fe aa14

- **`movq $-1, %rax`**

rax = ffff ffff ffff ffff

- **`movabsq $0x123456789ab, %rax`**

rax = 0000 0123 4567 89ab

- **`movq $-1, 0x4e0`**

Mem[0x004e4] = ffff ffff

Mem[0x004e0] = ffff ffff

## Istruzioni Più Comuni - Esempi `mov` - 3 - Zero/Signed

- Sintassi: **`mov[b,w,l,q]`** src, dst
- Condizioni iniziali:

Mem[0x00204] = 7654 3210

Mem[0x00200] = fedc ba98

rdx = 0123 4567 89ab cdef

- **`movslq`** 0x200, %rax
- **`movzwl`** 0x202, %eax
- **`movsbw`** 0x201, %ax
- **`movsbl`** 0x206, %eax
- **`movzbq`** %dl, %rax

rax = ffff ffff fedc ba98

rax = ffff ffff 0000 fedc

rax = ffff ffff 0000 ffba

rax = ffff ffff 0000 0054

rax = 0000 0000 0000 00ef

## Istruzioni Più Comuni - Esempi add, and, or, sub

- Sintassi: **mov[b,w,l,q]** src, dst
- Condizioni iniziali:

Mem[0x00204] = 7654 3210

Mem[0x00200] = 0f0f ff00

rdx = ffff ffff 1234 5678

rax = 0000 0000 cc33 aa55

- **addl** \$0x12300, %eax
- **addq** %rdx, %rax
- **andw** 0x200, %ax
- **orb** 0x203, %al
- **subw** \$14, %ax
- **addl** \$0x12345, 0x204

rax = 0000 0000 cc34 cd55

rax = ffff ffff de69 23cd

rax = ffff ffff de69 2300

rax = ffff ffff de69 230f

rax = ffff ffff de69 2301

Mem[0x00204] = 7655 5555

Mem[0x004e0] = 0f0f ff00

# Load Effective Address

- `lea`: load effective address
- Sintassi: **lea** src, dest
  - Istruzione “strana”, ma talvolta molto utile!
  - Nata per calcolare indirizzi (con indirizzamento indiretto) senza fare accessi
    - Usa l’hardware del calcolo di indirizzamento per “normali” operazioni aritmetiche
- Copia indirizzo di sorgente (calcolato tramite displacement, base, indice e scala) in registro destinazione
  - Calcola l’indirizzo e lo memorizza nel registro di destinazione senza “caricare” nulla dalla memoria.
  - Esempio:
    - **lea** 80(%rdx, %rcx, 2), %rax  $\rightarrow$  `%rax = %rdx + 2 * %rcx + 80`
  - Viene spesso utilizzata come istruzione aritmetica che effettua contemporaneamente due somme (un valore immediato e due registri) shiftando uno degli addendi.



## Load Effective Address - 2

Esempio: sommare due registri salvando risultato su un terzo registro

- Su RISC-V se vogliamo sommare `x1` e `x2` salvando il risultato in `x3`
  - **add** `x3, x1, x2`
- Su intel come possiamo sommare `%rbx` e `%rcx` salvando il risultato in `%rax`?
  - Non è possibile specificare un registro destinazione per `add` diverso dal secondo operando
  - Ma se consideriamo un accesso a memoria con base `%rbx` ed indice `%rcx`...
  - ...l'indirizzo acceduto sarebbe `%rbx + %rcx`
  - **lea** (`%rbx, %rcx`), `%rax`
- L'istruzione `lea` è spesso utilizzata per fare più somme (con eventuali shift) salvando il risultato in un registro diverso

## Load Effective Address - Esempio

- Sintassi: **lea** src, dst
- Condizioni iniziali:

```
rcx = 0000 0000 0000 0020  
rdx = 0000 0089 1234 4000  
rbx = ffff ffff ff00 0300
```

- **leal** (%edx,%ecx),%eax
- **leaq** -8(%rbx),%rax
- **leaq** 12(%rdx,%rcx,2),%rax

```
rax = 0000 0000 1234 4020  
rax = ffff ffff ff00 02f8  
rax = 0000 0089 1234 404c
```

## Load Effective Address - Esempio 2

```
void fl(int x) { // x = %edi
    return 9 * x + 1;
}
```



Codice non ottimizzato:

```
fl:
    movl %edi, %eax    # tmp = x
    sall 3, %eax       # tmp *= 8
    addl %edi, %eax    # tmp += x
    addl $1, %eax      # tmp += 1
    ret
```

Codice ottimizzato:

```
fl:
    leal 1(%edi,%edi, 8), %eax
    # eax = 1 + %edi + %edi * 8
    ret
```

# Istruzioni Apparentemente Inutili

- `inc <register> / dec <register>`: somma / sottrae 1 a registro
  - A che servono? Perché non `add $1, <register>`?
    - In RISC-V e ARM ogni istruzione è codificata su 32 bit con numero costante di bit
    - Intel usa codifica binaria con numero variabile di bit
  - `add $1, <register>` richiederebbe di codificare:
    - Il codice operativo dell'istruzione `add`
    - Il valore immediato 1 (16, 32 o 64 bit)
    - Il registro `<register>`
  - `inc` non richiede di codificare il valore immediato
    - salvo almeno 16 bit!!!
  - Questo può spiegare il proliferare di istruzioni “apparentemente” inutili...

## Esempio

- Stringa C: array di caratteri terminato da 0
  - ASCII: caratteri codificati su byte
- Copia di una stringa:

```
void copia_stringa(char *d, const char *s)
{
    int i = 0;

    while ((d[i] = s[i]) != 0) {
        i += 1;
    }
}
```

- Esempio già visto per Assembly RISC-V...
- Come fare con Assembly Intel (64 bit)?

## Accesso a Singoli Byte

- Ricordate? Necessità di copiare **byte**, non **parole**...
- Intel fornisce soluzione semplice (ed elegante? Dipende dai punti di vista!)
  - I registri `rax ... rdx` sono composti da “sottoregistri”
  - In particolare, `al ... dl`: registri a 8 bit
- Operazioni **mov** da e a memoria possono lavorare su dati di diversa ampiezza (8, 16, 32 e 64 bit)
  - Istruzione `movb`: da memoria a `al .. dl` o viceversa
- Non c'è bisogno di una istruzione diversa come per RISC-V, è sempre la stessa `mov` che lavora su “parti” diverse del registro

## Implementazione Assembly: Prologo

- I due parametri `d` ed `s` sono contenuti in `%rdi` e `%rsi`
- Supponiamo di usare `%rax` per il contatore `i`
  - Non è un registro preservato...
  - Non è necessario salvarlo sullo stack
- Non è necessario alcun prologo; possiamo cominciare col codice C
- Iniziamo: `i = 0; → %rax = 0`

```
movq $0, %rax
```

## Implementazione Assembly: Loop

- Ciclo `while`: copia `s[i]` in `d[i]`
  - Prima di tutto, carichiamolo in `%bl`
  - Per fare questo, possiamo sfruttare la modalità di indirizzamento indiretta ( $\text{base} + \text{indice} * 2^{\text{scala}}$ , con  $\text{scala} = 0$ )
  - Nessuna necessità di caricare l'indirizzo dell'`i`-esimo elemento in un registro, come si faceva per RISC-V

```
L1: movb (%rsi, %rax), %bl    # Inizio Loop
```

- Ora memorizziamo `%bl` in `d[i]`

```
movb %bl, (%rdi, %rax)
```



## Implementazione Assembly: Fine del Loop

- Bisogna ora controllare se  $s[i] == 0$ 
  - Se sì, fine del loop!

```
cmpb $0, %bl    # confronta %bl con 0...  
je L2           # se sono uguali, salta a L2  
                # (esci dal loop!)
```

- Se no, incrementa  $i$  (che sta in `%rax`) e cicla...

```
add $1, %rax  
jmp L1           # L1: label di inizio loop
```

- La label `L2` implementerà il ritorno al chiamante

# Implementazione Assembly: Fine

- Non abbiamo salvato nulla sullo stack: non c'è necessità di epilogo!
- Si può direttamente tornare al chiamante

**L2: ret**

- Mettendo tutto assieme:

```
.text
.globl copia_stringa
copia_stringa:
    movq $0, %rax
L1:
    movb (%rsi, %rax), %bl
    movb %bl, (%rdi, %rax)
    cmpb $0, %bl
    je L2
    add $1, %rax
    jmp L1
L2:
    ret
```



# Esempi - codice

Codice	Assembler	Assunzioni/Note
<pre>int x, y, z; ... z = x + y;</pre>	<pre><b>movl</b> \$0x10000004, %ecx <b>movl</b> (%ecx), %eax <b>addl</b> 4(%ecx), %eax <b>movl</b> %eax, 8(%ecx)</pre>	<pre>&amp;x = 0x10000004 &amp;y = 0x10000008 &amp;z = 0x1000000c</pre>
<pre>char a[100]; ... a[1]--;</pre>	<pre><b>movl</b> \$0x1000000c, %ecx <b>decb</b> 1(%ecx)</pre>	<pre>&amp;a = 0x1000000c</pre>
<pre>int d[4], x; ... x = d[0]; x += d[1];</pre>	<pre><b>movl</b> \$0x10000010, %ecx <b>movl</b> (%ecx), %eax <b>movl</b> %eax, 16(%ecx) <b>movl</b> 16(%ecx), %eax <b>addl</b> 4(%ecx), %eax <b>movl</b> %eax, 16(%ecx)</pre>	<pre>&amp;d = 0x10000010 &amp;x = 0x10000020</pre>
<pre>unsigned int y; short z; y = y/4; z = z &lt;&lt; 3;</pre>	<pre><b>movl</b> \$0x10000010, %ecx <b>movl</b> (%ecx), %eax <b>shrl</b> 2, %eax <b>movl</b> %eax, (%ecx) <b>movw</b> 4(%ecx), %ax <b>salw</b> 3, %ax <b>movw</b> %ax, 4(%ecx)</pre>	<pre>&amp;y = 0x10000010 &amp;z = 0x10000014</pre>

## Esempi - codice 2

```
// data = %edi
// val  = %esi
// i    = %edx
int func_1(int data[], int *val, int i) {
    int sum = *val;
    sum += data[i];
    return sum;
}
```

```
struct Data {
    char c; int d;
}
// ptr = %edi
// x   = %esi
void func_2(struct Data* ptr, int x) {
    ptr->c++;
    ptr->d -= x;
}
// ptr = %edi
// x   = %esi
```

```
func_1:
    movl    (%esi), %eax
    addl    (%edi, %edx, 4), %eax
    ret
```

```
func_2:
    addb    $1, (%edi)
    subl    %esi, 4(%edi)
    ret
```

La convenzione per X86\_64 richiede che il valore di ritorno di una funzione sia memorizzato in `%eax/%rax`

## Esempi - codice 3

```
void abs_value(int x, int * res) {  
    if (x < 0)  
        *res = -x;  
    else  
        *res = x;  
}
```

```
// x = %edi, y = %esi, res = %rdx  
void func_3(int x, int y, int * res) {  
    if (x < y)  
        *res = x;  
    else  
        *res = y;  
}
```

```
abs_value: test %edi, %edi  
           jns Lab1  
           negl %edi  
           movl %edi, (%rsi)  
           ret  
Lab1:      movl %edi, (%rsi)  
           ret
```

```
func_3:  
          cmpl %esi, %edi  
          jge Lab2  
          movl %edi, (%rdx)  
          ret  
Lab2:     movl %esi, (%rdx)  
          ret
```

## Esempi - codice 4

```
// x = %edi, y = %esi, res = %rdx
void func_4(int x, int y, int * res) {
    if ((x == -1) || (y == -1))
        *res = y - 1;
    else if ((x > 0) && (y < x))
        *res = x + 1;
    else
        *res = 0;
}
```

```
// a = %edi, b = %esi
int avg(int a, int b) {
    return (a+b)/2;
}
```

```
func_4:
    cmpl $-1, %edi
    je .L6
    cmpl $-1, %esi
    je .L6
    testl %edi, %edi
    jle .L5
    cmpl %esi, %edi
    jle .L5
    addl $1, %edi
    movl %edi, (%rdx)
    ret

.L5:
    movl $0, (%rdx)
    ret

.L6:
    subl $1, %esi
    movl %esi, (%rdx)
    ret
```

```
avg:    movl %edi, %eax
        addl %esi, %eax
        sarl 1, %eax
        ret
```

## Esempi - codice 5

```
// str = %rdi
int func_5(char str[]) {
    int i = 0;
    while(str[i] != 0){
        i++;
    }
    return i;
}
```

```
// dat = %rdi, len = %esi
int func_6(int dat[], int len) {
    int min = dat[0];
    for (int i=1; i < len; i++) {
        if (dat[i] < min) {
            min = dat[i];
        }
    }
    return min;
}
```

```
func_5:
    movl $0, %eax
    jmp .L2
.L3:
    addl $1, %eax
.L2:
    movslq %eax, %rdx
    cmpb $0, (%rdi,%rdx)
    jne .L3
    ret
```

```
func_6:
    movl (%rdi), %eax
    movl $1, %edx
    jmp .L2
.L4:
    movslq %edx, %rcx
    movl (%rdi,%rcx,4), %ecx
    cmpl %ecx, %eax
    jle .L3
    movl %ecx, %eax
.L3:
    addl $1, %edx
.L2:
    cmpl %esi, %edx
    jl .L4
    ret
```



## Esempi - codice 6 - elevato numero parametri

```
int caller() {  
    int sum = f1(1, 2, 3, 4, 5, 6, 7, 8);  
    return sum;  
}  
  
int f1(int a1, int a2, int a3, int a4,  
       int a5, int a6, int a7, int a8) {  
    return a1+a2+a3+a4+a5+a6+a7+a8;  
}
```

```
caller:  
    pushq $8  
    pushq $7  
    movl $6, %r9d  
    movl $5, %r8d  
    movl $4, %ecx  
    movl $3, %edx  
    movl $2, %esi  
    movl $1, %edi  
    call f1  
    addq $16, %rsp  
    ret  
  
f1:  
    addl %edi, %esi  
    addl %esi, %edx  
    addl %edx, %ecx  
    addl %ecx, %r8d  
    addl %r8d, %r9d  
    movl %r9d, %eax  
    addl 8(%rsp), %eax  
    addl 16(%rsp), %eax  
    ret
```