SAPIENZA
UNIVERSITÀ DI ROMA

Sistemi di calcolo

Capitolo 3: Parte III
Come viene tradotto in assembly un programma C?

Corso di Laurea in Ingegneria Informatica e Automatica

Conversioni di tipi numerici

a=c	char c	short c	int c
char a	movb %cl, %al		
short a	movsbw %cl, %ax	movw %cx, %ax	
int a	movsbl %cl, %eax	movswl %cx, %eax	movl %ecx, %eax

movs: sorgente non immediata, destinazione solo registro

Conversioni di tipi numerici

Esempi:

%cl = 0x80

movsbl %cl, %eax ⇒ eax = 0xFFFFFF80

propagazione bit del segno ←

%cl = 0x40

movsbl %cl, %eax ⇒ eax = 0x00000040

s → signed

← propagazione bit del segno

Conversioni di tipi numerici

a=c	sign. o uns. char c	sign. o uns. short c	sign. o uns. int c
uns. char a	movb %cl, %al		
uns. short a	movzbw %cl, %ax	movw %cx, %ax	
uns. int a	movzbl %cl, %eax	movzwl %cx, %eax	movl %ecx, %eax

movz: stessi vincoli sugli operandi di movs

Conversioni di tipi numerici

Esempio:

%cl = 0x80

movzbl %cl, %eax ⇒ eax = 0x00000080

zero

completa con zeri

Conversioni di tipi numerici

DEMO 3.8-cast

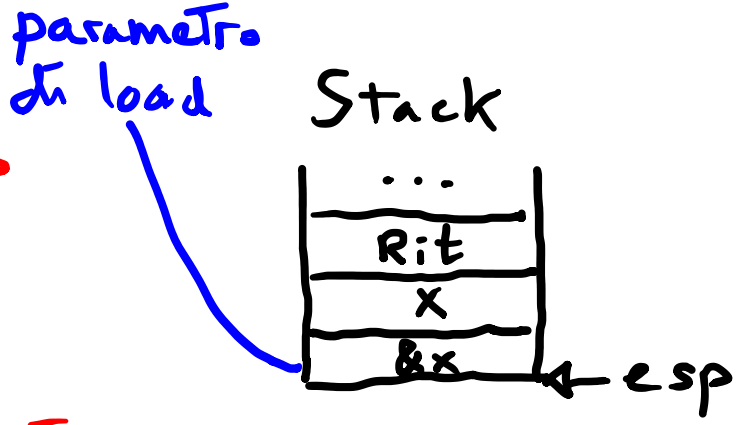
Variabili locali in stack

Finora abbiamo sempre tenuto le variabili locali in registri.

Motivi per non farlo:

1) più variabili locali che registri
2) si richiede l'indirizzo di una variabile &x. Es:

```
int f() {
    int x;
    load(&x);
    return x;
}
```




```
f: subl $8,%esp
   movl %esp,%ecx
   addl $4,%ecx
   movl %ecx,(%esp)
   call load
   movl 4(%esp),%eax
   addl $8,%esp
   ret
```

Calcolo espressioni mediante istruzione lea

L'operando memoria più complesso ha la forma:

d (base, indice, scala) che denota l'indirizzo

d + base + indice * scala, dove { d = costante; base, indice = registri; scala = 1, 2, 4, 8

Solitamente l'operando viene utilizzato per accedere all'indirizzo.

L'istruzione lea calcola l'indirizzo, ma non vi accede.

Es. leal 4(%esp), %ecx è equivalente a { movl %esp, %ecx; addl $4, %ecx

NB: lea non modifica il registro EFLAGS

Calcolo espressioni mediante istruzione lea

Es. `leal -4(%eax), %ecx` ⟷ `ecx = eax - 4`

`movl -4(%eax), %ecx` ⟷ `ecx = *(eax - 4)`

Es. `leal (%eax), %ecx` ⟷ (?, sì) `movl %eax, %ecx`

Es. `leal 7(%eax, %ecx, 4), %edx` ⟷

```
movl %ecx, %edx
imull $4, %edx
addl %eax, %edx
addl $7, %edx
```

} tutte queste istruzioni sono rimpiazzabili da una sola lea

Calcolo espressioni mediante istruzione lea

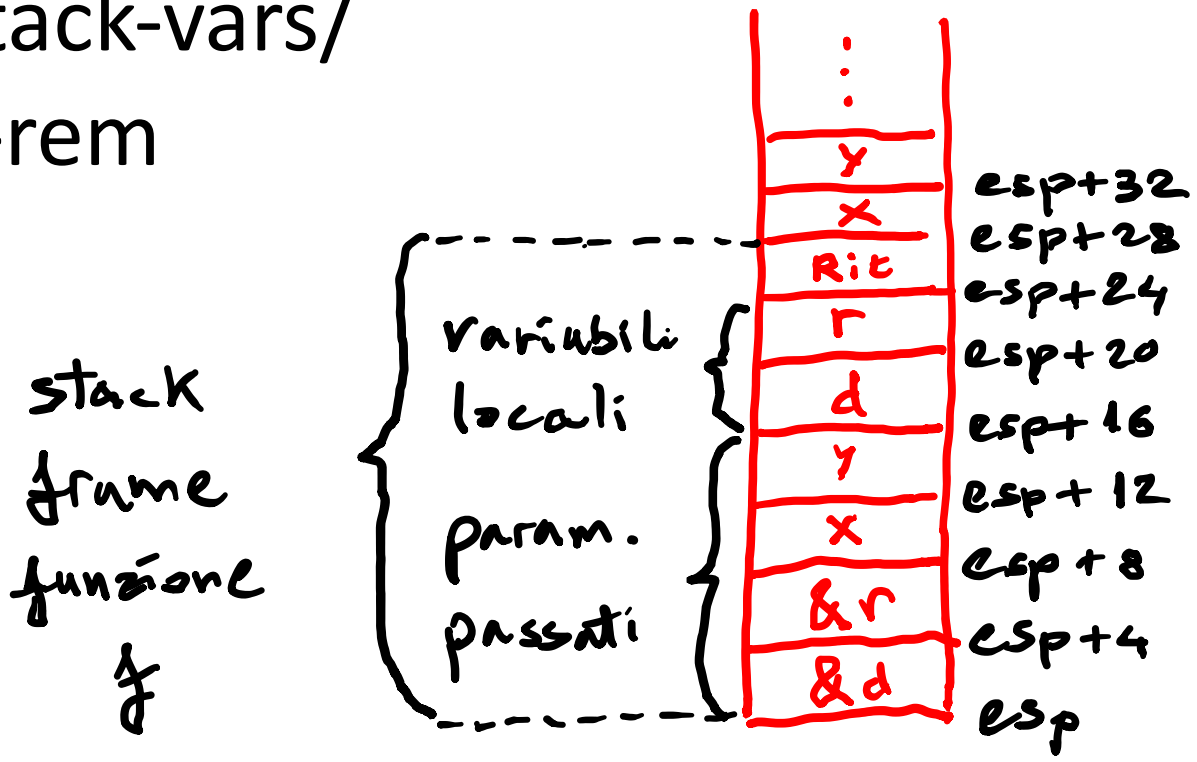
```
int f() {
    int x;
    load(&x);
    return x;
}
```

```
f: subl $8,%esp
   movl %esp,%ecx
   addl $4,%ecx
   movl %ecx,(%esp)
   call load
   movl 4(%esp),%eax
   addl $8,%esp
   ret
```

leal 4(%esp),%ecx

Variabili locali in stack

DEMO 3.9-stack-vars/
get-div-rem



Espressioni booleane

```
int f(int x, int y){
    return x<y;
}
```

soluzione "verbosa"

```
if (x<y) return 1;
return 0;
```

Esiste una soluzione più compatta che usa l'istruzione setcc:

condition code
e, ne, l, le, ecc.

```
f: movl 4(%esp), %ecx   #x
   movl 8(%esp), %edx   #y
   cmpl %edx, %ecx
   setl %al
   movzbl %al, %eax
   ret
```

Variabili locali in stack

DEMO 3.A-bool-expr

Cortocircuitazione

Attenzione alla regola di cortocircuitazione nelle espressioni booleane del C:

```
int stringa_vuota(const char * s){
   return s!=NULL && *s==0;
}
```

viene eseguita solo se s!=NULL è vera

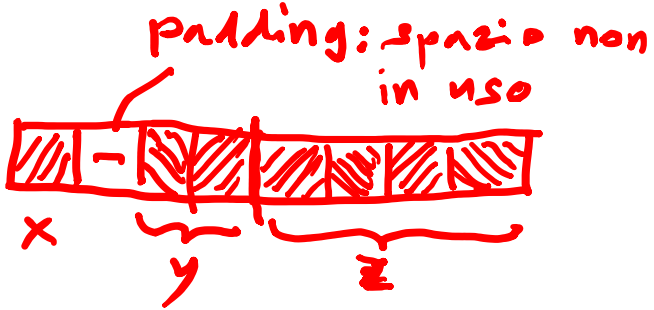
Consultazione manuale Intel

Instruction set reference, A-Z

https://software.intel.com/en-us/articles/intel-sdm

Strutture (struct): allineamento e padding

Per motivi di miglior prestazioni nei sistemi x86 un oggetto di x byte deve iniziare a un indirizzo multiplo di x. Es. struttura C:




```
struct S {
  char x;
  short y;
  int z;
};
```

La size di una struttura sarà multiplo della size dell'attributo più grande.

Strutture (struct): allineamento e padding

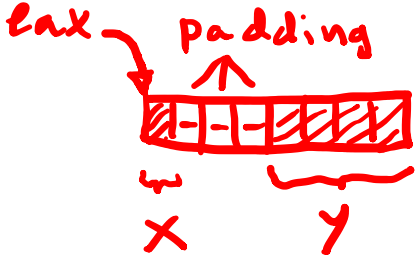
Esempio:

```
struct S {
    char x;
    int y;
};
```

```
void f(struct S* p, char x, int y) {
    p->x = x;
    p->y = y;
}
```

```
f: movl 4(%esp), %eax
   movl 8(%esp), %ecx
   movl 12(%esp), %edx
   movb %cl, (%eax)
   movl %edx, 4(%eax)
   ret
```

promozione intera



Liste collegate: somma degli elementi di una lista

```
typedef struct nodo nodo;
struct nodo {
  int elem;
  nodo *next;
};
int sum(nodo *p) {
  int cnt = 0;
  for (; p!=NULL; p=p->next)
    cnt += p->elem;
  return cnt;
}
```


```
.globl sum
sum: movl 4(%esp), %ecx
     xorl %eax, %eax
L:   testl %ecx, %ecx
     je E
     addl (%ecx), %eax
     movl 4(%ecx), %ecx
     jmp L
E:   ret
```

Strutture (struct):

DEMO 3.B-struct

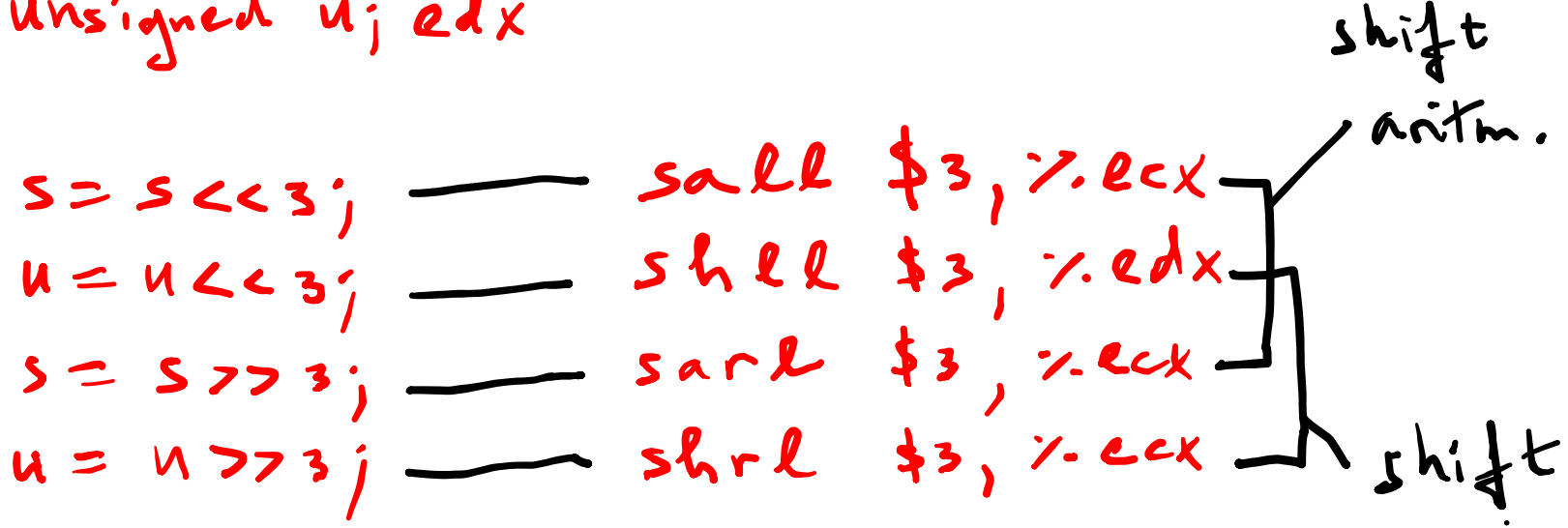
Istruzioni di shift

`int s;` ↔ `ecx`

`unsigned u;` `edx`

`x<<y` moltiplica `x` per 2^y

`x>>y` divide `x` per 2^y



Se manca il primo operando si intende shift di 1 bit

Istruzioni di shift

Es. int s = 0xABADCAFE ↔ ecx

unsigned u = 0xABADCAFE ↔ edx

sall $8, %ecx → ecx = 0xADCAFE00 } equivalenti

shll $8, %edx → edx = 0xADCAFE00 }

sarl $8, %ecx → ecx = 0xFFABADCA

shrl $8, %edx → edx = 0x00ABADCA

Istruzione idiv con dividendo negativo (uso shift)

idiv realizza la divisione e il resto tra il dividendo a 64 bit ottenuto concatenando i registri D e A e il divisore che può essere un registro arbitrario.

Dividendo

Divisione tra argomenti a 32 bit

Estensione segno.

```
movl %eax, %edx   } estensione
sarl $31, %edx    } segno.
idiv %ecx         — divisore
```

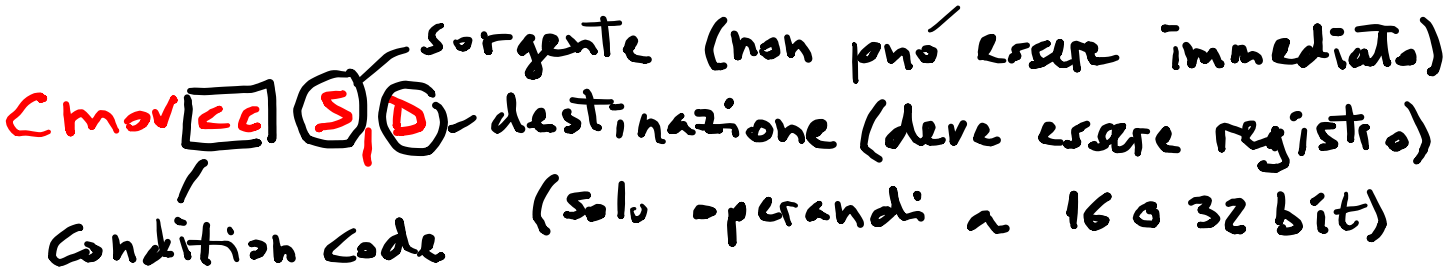
NB: il divisore può essere anche operando memoria.

Istruzioni di shift

DEMO 3.C-shift

Movimento dati condizionale: cmov

L'ISA IA32 fornisce un'istruzione mov condizionale che semplifica la realizzazione di costrutti if:



Es. `if (a>b) b=a;` ⇒

```
cmpl %ebx, %eax
cmovgl %eax, %ebx
```

Istruzioni di shift

DEMO 3.D-cmov

Ricapitolazione vincoli operandi

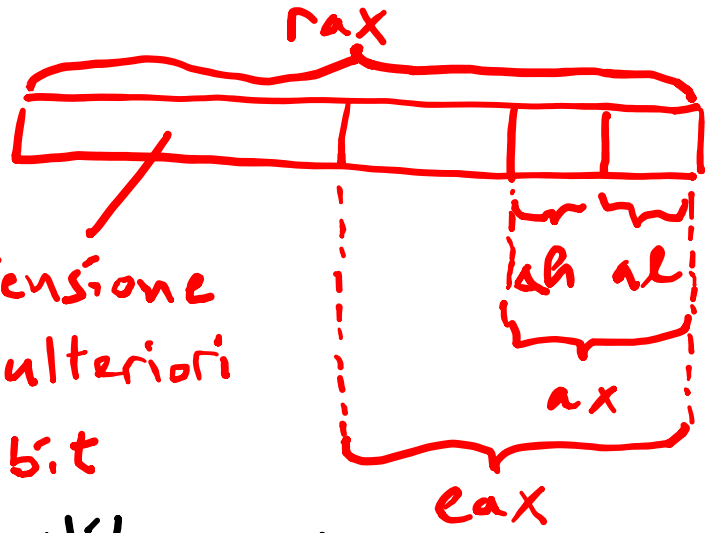
Istruzione	Vincolo	Esempio
IMUL	la destinazione deve essere un registro	`imull $2, %eax  # ok` `imull $2, (%edi)  # errore`
CMOVcc	la sorgente non può essere un immediato	`cmovgel %ecx, %eax  # ok` `cmovgel $2, %eax  # errore`
	la destinazione deve essere un registro	`cmovgel %eax, %eax  # ok` `cmovgel %eax, (%edi)  # errore`
	solo operandi 16 o 32 bit	`cmovgeb %cl, %al  # errore`
TEST e CMP	secondo operando ("destinazione") non può essere immediato	`testl %eax, %ecx  # ok` `testl (%eax), $2  # errore`

MOVZ e MOVS	sorgente non immediato e destinazione solo registro	`movzbl (%eax), %ecx  # ok`
		`movzbl $2, %ecx  # errore`
		`movzbl %eax, (%ecx)  # errore`

DIV	usa solo i registri D ed A e l'operando dividendo non può essere immediato	`movl %edi  # ok` `movl %1  # errore`

Estensione a x86-64: ISA

- Altri 8 registri a 64 bit: R8, R9, ..., R15
- Estensione a 64 bit dei registri visti



E' possibile accedere al byte meno signif. di DI e SI con dil e sil

Sufissi:

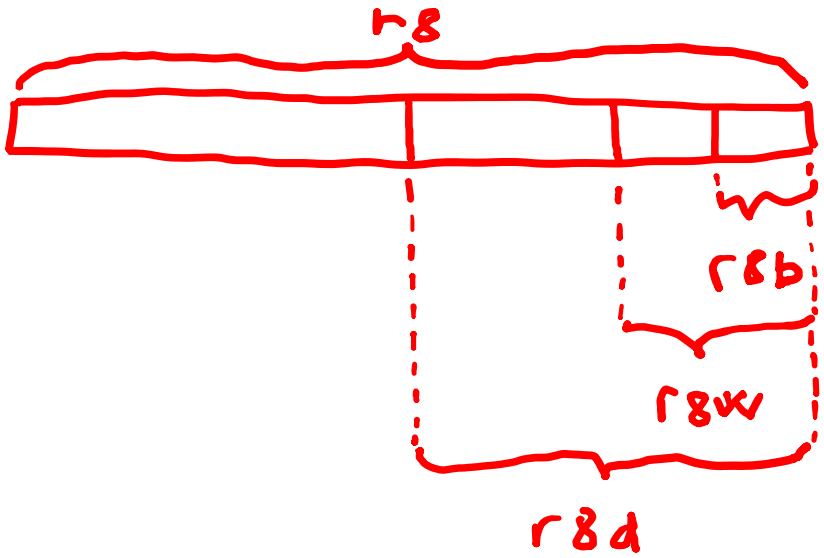
b	1 byte	char
w	2 byte	short
l	4 byte	int
q	8 byte	long, puntatori

Es. movq $10, %rax

Estensione a x86-64: ISA

Sottoparti dei registri R8–R15:

Es.:



Qualsiasi scrittura nei 32 bit meno significativi azzera i 32 bit più significativi.

Le istruzioni movs e movz possono essere usate in un contesto a 64 bit usando i suffissi b, w, l, q.

Es. movsbq %al, %rax

Estensione a x86-64: ABI

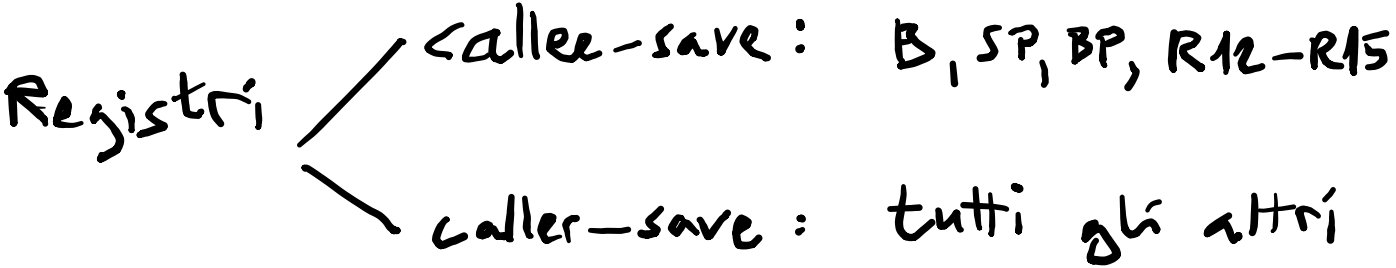
Il passaggio dei parametri avviene nei registri rdi, rsi, rdx, rcx, r8, r9 per i primi 6 parametri, e in stack per gli altri.

Es.

```
int swap(int*x, int*y){
    int t=*x;
    *x=*y;
    *y=t;
}
```

```
.globl swap
swap:
  movl (%rdi), %eax
  movl (%rsi), %ecx
  movl %eax, (%rsi)
  movl %ecx (%rdi)
  ret
```

Estensione a x86-64: ABI



Estensione a x86-64

DEMO 3.E-x86-64

Esercizi riepilogativi

DEMO 3.F-recap