Sistemi di Calcolo

Modulo 1:
Programmazione dei sistemi di calcolo a singolo nodo

Ultimo aggiornamento: 30 ottobre 2014

Camil Demetrescu
Dipartimento di Ingegneria Informatica, Automatica e Gestionale "A. Ruberti"
Sapienza Università di Roma

Indice

1 Cosa è un sistema di calcolo?

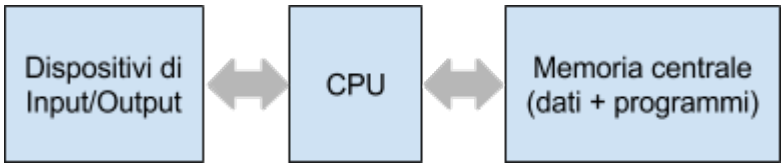
Un **sistema di calcolo** consiste di **software** e **hardware** che operano insieme per supportare l'esecuzione di programmi. Esempi di sistemi di calcolo sono gli smartphone, i tablet, i computer fissi e i portatili, i data center che gestiscono i nostri account Facebook, Twitter o Google, i supercomputer usati dal CERN di Ginevra per simulare la fisica delle particelle, ma anche i televisori di nuova generazione che consentono di navigare in Internet, i riproduttori multimediali, i modem/router che usiamo a casa per connetterci alla rete ADSL, le macchine fotografiche digitali, i computer di bordo delle automobili, le console per i videogiochi (PlayStation, Wii, Xbox, PS3, ecc.), e molto altro ancora che non sospetteremmo possa essere pensato come un sistema di calcolo.

In generale, un sistema di calcolo è qualsiasi **sistema programmabile**, cioè in grado di eseguire compiti diversi in base alle istruzioni fornite da un **programma**. Un sistema di calcolo può essere formato da un **singolo nodo**, cioè un insieme di parti hardware strettamente connesse tra loro e spazialmente adiacenti, oppure da **più nodi connessi mediante una rete di comunicazione**.

Sono sistemi di calcolo a singolo nodo i computer fissi e portatili, gli smartphone, i tablet, ecc. Esempi di sistemi multi-nodo sono i data center usati dai grandi provider come Facebook, Twitter e Google per gestire i loro sistemi di social networking e i supercomputer, in cui più nodi di calcolo sono connessi da una rete di comunicazione ad alta velocità (es. Infiniband). Questo tipo di sistema viene anche detto **cluster**.

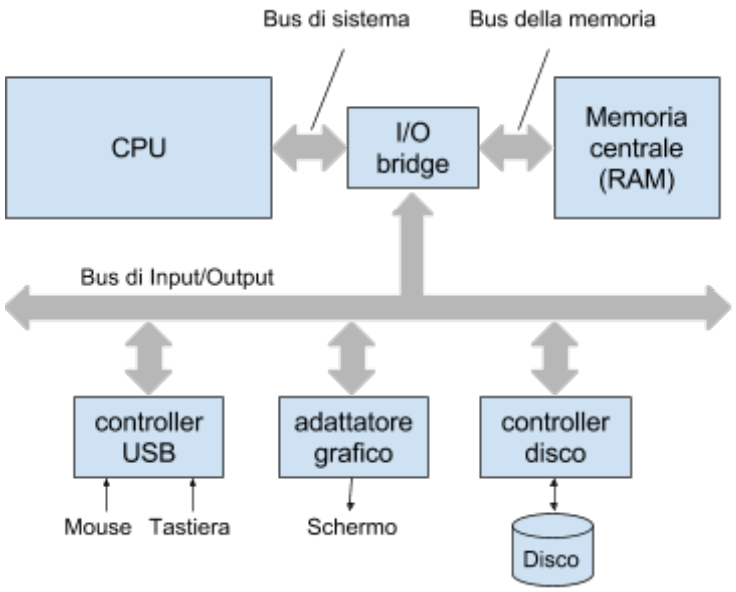
Nel primo modulo di corso tratteremo la programmazione dei sistemi a singolo nodo, mentre i sistemi multi-nodo saranno affrontati nel secondo modulo.

In questo corso tratteremo sistemi di calcolo in cui l'hardware dei singoli nodi è organizzato secondo il **modello di Von Neumann**:



La **memoria centrale** contiene **dati da elaborare** e **programmi**, i **dispositivi di input e output** scambiano dati e interagiscono con il mondo esterno, mentre la **CPU** (Central Processing Unit, o Unità Centrale di Elaborazione) esegue le **istruzioni** di un programma.

Più in dettaglio, l'organizzazione tipica di un calcolatore moderno è la seguente:



Il diagramma fornisce in maggiore dettaglio le componenti architetturali tipiche di un sistema di calcolo a singolo nodo, descritte nei paragrafi seguenti.

1.1 CPU

La **CPU** (o **microprocessore**) è il cuore del sistema che esegue programmi scritti in **linguaggio macchina** (codice **nativo**), rappresentati come sequenze di byte che codificano istruzioni. Il **set di istruzioni**, cioè l'insieme delle istruzioni riconosciute dalla CPU è specifico alla particolare famiglia di CPU, e può differire anche sostanzialmente fra modelli diversi prodotti da aziende diverse. Le istruzioni vengono eseguite dalla CPU nell'ordine in cui appaiono in memoria ed effettuano tipicamente operazioni che:

A. **trasferiscono dati** all'interno della CPU, all'interno della memoria, fra memoria e CPU, e fra dispositivi esterni e la CPU.
B. calcolano operatori **aritmetico/logici** (somme, prodotti, ecc.), operatori **booleani** (congiunzione, disgiunzione, negazione, ecc.), operatori **relazionali** (uguale, maggiore, minore, ecc.).
C. effettuano **salti** che permettono di continuare l'esecuzione non dall'istruzione successiva in memoria, ma da un altro punto del programma. Queste istruzioni servono per realizzare cicli (for, while, ecc.) e costrutti di selezione (if … else).

1.2 Bus

Sono strutture di interconnessione che collegano le varie componenti del sistema consentendo lo scambio dei dati. I bus sono canali di comunicazione che trasferiscono i dati tipicamente a blocchi fissi di byte, chiamati **word**. La dimensione di una word trasferita su un bus è generalmente di 4 byte (32 bit) oppure 8 byte (64 bit), a seconda dell'architettura. In questa dispensa considereremo word di 8 byte (architettura a 64 bit) e assumeremo che su un bus viene trasferita una word alla volta.

1.3 I/O bridge

Si occupa di coordinare lo scambio dei dati fra la CPU e il resto del sistema.

1.4 Controller e adattatori

Interfacciano il sistema verso il mondo esterno, ad esempio acquisendo i movimenti del mouse o i tasti premuti sulla tastiera. La differenza fra i controller e adattatore è che i controller sono saldati sulla **scheda madre** (cioè sul circuito stampato che ospita CPU e memoria e bus) oppure sono integrati nel dispositivo esterno, mentre gli adattatori sono schede esterne collegate alla scheda madre (es. adattatore video o di rete).

1.5 Domande riepilogative

Domande

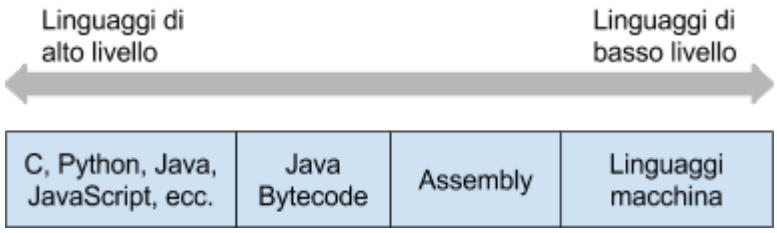
---

2 Come viene programmato un sistema di calcolo?

2.1 Linguaggi di programmazione di alto livello e di basso livello

Il linguaggio macchina è pensato per essere eseguito dalla CPU, ma è assolutamente inadatto per programmare. I programmatori scrivono invece i loro programmi in **linguaggi di alto livello** come C, Python, Java, ecc., che forniscono costrutti molto più potenti, sono più semplici da imparare e usare, sono più facilmente manutenibili, ed è più semplice identificare e correggere gli errori (**debugging**).

I programmatori professionisti interessati a scrivere codice particolarmente ottimizzato per le prestazioni oppure parti di sistemi operativi usano il linguaggio **assembly** (o assemblativo), un linguaggio di basso livello che descrive le istruzioni macchina utilizzando una sintassi comprensibile allo sviluppatore. I linguaggi assembly sono "traslitterazioni" dei corrispondenti linguaggi macchina che associano a ogni codice di istruzione binario un corrispondente codice mnemonico, più leggibile per un programmatore.



2.2 Compilazione vs. interpretazione

Per poter eseguire un programma scritto in un linguaggio di alto livello ci sono vari approcci possibili:

Approccio	**Compilazione**	**Interpretazione**	**Ibrido: compilazione e interpretazione**
Descrizione	Il programma di alto livello viene **tradotto in codice macchina** (nativo) in modo da poter essere eseguito direttamente dalla CPU. In generale, il processo di	Il programma di alto livello viene direttamente eseguito da un programma chiamato **interprete**, senza bisogno di essere prima compilato. I linguaggi di alto	Il programma di alto livello viene prima compilato in un codice scritto in un linguaggio di livello **intermedio**. Il programma di livello intermedio viene poi

	traduzione da un linguaggio all'altro viene chiamato **compilazione**.	livello interpretati vengono generalmente chiamati **linguaggi di scripting**.	interpretato.
Esempi linguaggi	C, C++, Fortran	Python, Perl, PHP, JavaScript	Java
Vantaggi	Il compilatore genera **codice ottimizzato** per la piattaforma di calcolo considerata, ottenendo le migliori **prestazioni** possibili per un programma di alto livello.	Il programma è **portabile**, essendo eseguibile su qualsiasi piattaforma su cui sia disponibile un interprete per il linguaggio in cui è scritto. Inoltre, è possibile **eseguirlo direttamente** senza bisogno di compilarlo, rendendo più agile lo sviluppo.	Il programma è **portabile**, essendo eseguibile su qualsiasi piattaforma su cui sia disponibile un interprete per il linguaggio intermedio in cui è stato compilato.
Svantaggi	Il programma compilato **non è portabile**, essendo eseguibile solo sulla piattaforma di calcolo per cui è stato compilato	L'esecuzione interpretata è tipicamente più **lenta** di quella nativa ottenuta mediante un compilatore che genera codice macchina.	Vedi interpretazione.

2.1 Stadi della compilazione di un programma C

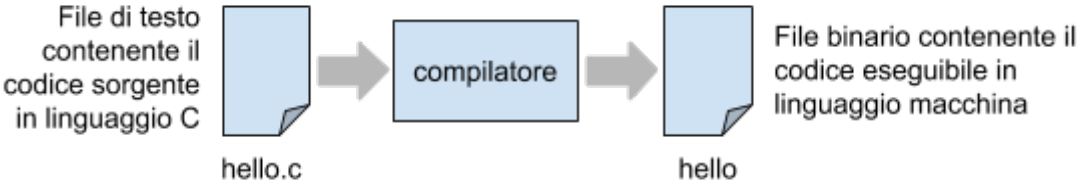
La compilazione di un programma C può essere scomposta nella compilazione di più moduli C (anche dette "translation unit"), ciascuno residente in un file di testo con estensione ".c". Nei casi più semplici, un programma C è formato da un'unica translation unit:

hello.c

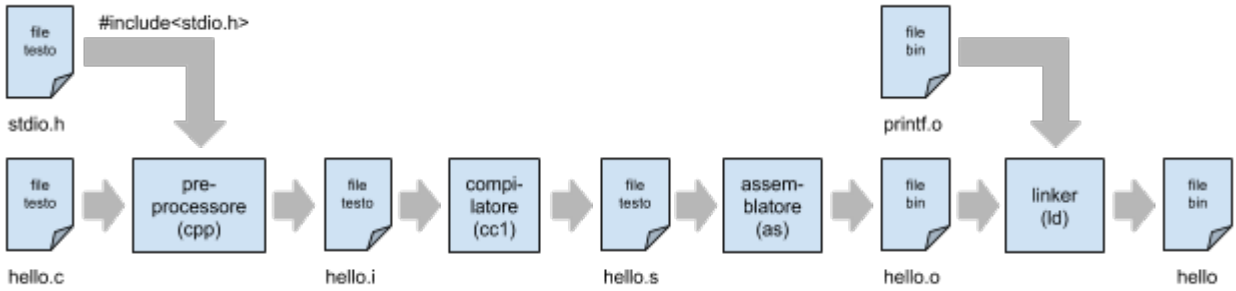
```
#include<stdio.h>

int main() {
   printf("Hello world\n");
   return 0;
}
```

La compilazione parte dal file `hello.c` e genera un file eseguibile hello:



Il processo di compilazione di un programma C è in realtà composto da diversi stadi, che vengono normalmente effettuati dal sistema di compilazione (*compilation toolchain*, in inglese) senza che ce ne accorgiamo:



Gli stadi coinvolgono diversi sottoprogrammi che vengono attivati dal sistema di compilazione:

1. **Preprocessore**: prende un testo C e lo converte in un altro testo C dove le direttive `#include`, `#define`, ecc. sono state elaborate. Il testo risultante sarà un programma C senza le direttive `#`. Il preprocessore includerà nel file generato il contenuto di tutti i file che vengono specificati dalle direttive `#include`. Nel nostro esempio, oltre a leggere `hello.c`, il preprocessore leggerà anche il file `stdio.h`, disponibile nelle directory di sistema del compilatore.
2. **Compilatore**: prende un testo C senza direttive `#`, ne verifica la correttezza sintattica, e lo traduce in codice in linguaggio assembly per la piattaforma per cui si sta compilando.
3. **Assemblatore**: prende un programma scritto in assembly e lo traduce in codice macchina, generando un **file oggetto** (binario).
4. **Linker**: prende vari file oggetto e li collega insieme a formare un unico file eseguibile. Nel nostro esempio, verranno collegati ("linkati") insieme `hello.o`, che contiene il programma in codice macchina, e `printf.o`, che contiene il codice macchina che realizza la funzione `printf` invocata dal programma. Il risultato della compilazione è il file eseguibile `hello`.

2.1.1 Il toolchain di compilazione `gcc`

Linux/MacOS X

Il toolchain di compilazione `gcc` contiene vari programmi che realizzano i diversi stadi della compilazione:

1. `cpp`: preprocessore
2. `cc1`: compilatore C
3. `as`: assemblatore
4. `ld`: linker

Normalmente non si invocano questi programmi direttamente, ma i vari stadi possono essere effettuati separatamente mediante l'inclusione di opportune opzioni ("switch" della forma `-`*`option`*, dove *`option`* può essere: `E`, `S`, `c`, `o`) nella riga di comando con cui si invoca `gcc`:

1. `gcc -E hello.c > hello.i`: preprocessa `hello.c` e genera il programma C preprocessato `hello.i`, in cui le direttive `#` sono state elaborate. Si noti che `gcc -E hello.c` stamperebbe il testo preprocessato a video. L'uso della redirezione `> hello.i` dirotta invece l'output del preprocessore nel file `hello.i`.
2. `gcc -S hello.i`: compila il programma preprocessato `hello.i` traducendolo in un file assembly `hello.s`.
3. `gcc -c hello.s`: assembla il file `hello.s`, scritto in linguaggio assemby, generando un file oggetto `hello.o`.
4. `gcc hello.o -o hello`: collega il file oggetto `hello.o` con i moduli della libreria di sistema (ad esempio quella contenente il codice della funzione `printf)` e genera il file eseguibile `hello`.

I passi elencati ai punti 1-4 sopra generano il file eseguibile `hello` a partire da un file sorgente `hello.c` creando uno ad uno tutti i file relativi ai vari stadi intermedi della compilazione (`hello.i`, `hello.s`, `hello.o`). Si noti che il comando:

```
gcc hello.c -o hello
```

è del tutto equivalente, con l'unica differenza che i file intermedi vengono creati come file temporanei nascosti al programmatore e poi eliminati automaticamente da `gcc`.

2.1.2 Programmi in formato testo, oggetto, ed eseguibile (ELF, Mach-O)

Linux/MacOS X

Si può esaminare la natura dei vari file coinvolti nel processo di compilazione usando il comano `file`. Questo è l'output che si otterrebbe su sistema operativo Linux a 64 bit:

```
$ file hello.c
hello.c: C source, ASCII text

$ file hello.i
hello.i: C source, ASCII text

$ file hello.s
hello.s: ASCII text
```

```
$ file hello.o
hello.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV),
not stripped

$ file hello
hello: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.24,
BuildID[sha1]=0x6c7467509edcd8c76549721c01468b480a6988f4, not
stripped
```

Si noti che i file `hello.c`, `hello.i` e `hello.s` sono file di testo (ASCII text), mentre i file `hello.o` e `hello` sono file binari. In particolare, gli ultimi due usano il formato binario ELF che rappresenta tutte le informazioni di un programma in codice macchina. Il formato ELF è usato da numerosi altri sistemi oltre a Linux, fra cui le console PlayStation e alcuni smartphone.

Il sistema operativo MacOS X (ma anche iOS, usato su iPhone e iPad) usa invece il formato binario Mach-O per rappresentare file oggetto ed eseguibili. Il seguente risultato è ottenuto su sistema operativo MacOS X a 64 bit:

```
$ file hello.c
hello.c: ASCII c program text

$ file hello.i
hello.i: ASCII c program text

$ file hello.s
hello.s: ASCII assembler program text

$ file hello.o
hello.o: Mach-O 64-bit object x86_64

$ file hello
hello: Mach-O 64-bit executable x86_64
```

Vi sono numerosi altri formati per i file oggetto ed eseguibili dipendenti dalla particolare piattaforma utilizzata.

Si noti che un file in formato ELF non può essere eseguito o linkato su MacOS X. Allo stesso modo, un file in formato Mach-O non può essere eseguito o linkato in Linux. Tutto questo indipendentemente dall'hardware soggiacente, che potrebbe anche essere lo stesso. Pertanto, **i formati eseguibili non garantiscono in genere la portabilità** dei programmi.

2.1.2 Disassemblare programmi in codice macchina: `objdump`

Disassemblare un programma in **linguaggio macchina** consiste nel tradurne le istruzioni nel codice **assembly** corrispondente in modo che possano essere analizzate da un programmatore. E' l'operazione inversa a quella dell'assemblamento.

Linux: `objdump -d` **(disassemblato)**

In ambiente Linux è possibile disassemblare un file oggetto o un file eseguibile usando il comando `objdump -d`.

Consideriamo il seguente semplice modulo `somma.c`:

```
int somma(int x, int y) {
    return x + y;
}
```

Compilandolo otteniamo un file oggetto `somma.o`:

```
$ gcc -c somma.c
```

che possiamo disassemblare come segue:

```
$ objdump -d somma.o
```

Il comando `objdump` invia l'output sul canale standard (di default è il terminale):

```
somma.o:   file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <somma>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   89 7d fc                mov    %edi,-0x4(%rbp)
   7:   89 75 f8                mov    %esi,-0x8(%rbp)
   a:   8b 45 f8                mov    -0x8(%rbp),%eax
   d:   8b 55 fc                mov    -0x4(%rbp),%edx
  10:   01 d0                   add    %edx,%eax
  12:   5d                      pop    %rbp
  13:   c3                      retq
```

Si noti che l'output riporta per ogni funzione (in questo caso la sola `somma`):

1. lo spiazzamento (offset) dell'istruzione all'interno della sezione (in esadecimale)
2. il codice macchina (in esadecimale)
3. il corrispondente codice assembly (usando i nomi mnemonici delle istruzioni).

Nel nostro esempio, il codice macchina è della famiglia x86 e come si vede le istruzioni occupano un numero variabile di byte (ad esempio, il codice macchina dell'istruzione assembly `retq` è il byte `c3` e quello dell'istruzione assembly `mov %rsp,%rbp` sono i tre byte `48`, `89` ed `e5`).

Un **disassemblato misto** contiene il codice sorgente inframmezzato a quello macchina/assembly, facilitando al programmatore l'analisi del codice.

Linux: `objdump -S` **(disassemblato misto)**

Se il programma è stato compilato con l'opzione `-g` di `gcc`, usando il comando `objdump -S` è possibile ottenere un **disassemblato misto** in cui il codice sorgente e quello macchina/assembly sono inframmezzati:

```
$ gcc -g -c somma.c
$ objdump -S somma.o

somma.o:   file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <somma>:
int somma(int x, int y){
   0:    55                       push   %rbp
   1:    48 89 e5                 mov  %rsp,%rbp
   4:    89 7d fc                 mov  %edi,-0x4(%rbp)
   7:    89 75 f8                 mov  %esi,-0x8(%rbp)
    return x+y;
   a:    8b 45 f8                 mov  -0x8(%rbp),%eax
   d:    8b 55 fc                 mov  -0x4(%rbp),%edx
  10:    01 d0                    add  %edx,%eax
}
  12:    5d                       pop  %rbp
  13:    c3                       retq
```

Ad esempio, le istruzioni macchina/assembly con offset compreso tra `a` e `12` (escluso) sono la traduzione dell'istruzione `return x+y` del programma C.

---

3 Come vengono eseguiti i programmi?

3.1 Processi

Quando un processo viene eseguito, esso dà luogo a un processo. Un **processo** è semplicemente un **programma in esecuzione**.

Uno **stesso programma** può essere istanziato in **più processi** che possono coesistere nel sistema. Ogni processo è identificato univocamente dal un **identificatore di processo** chiamato **PID** (Process ID). Il PID è un **numero progressivo** che viene incrementato di uno ogni volta che viene creato un

nuovo processo.

Un processo è caratterizzato da principalmente da:

- Un'**immagine di memoria** che contiene il codice del programma e i dati da esso manipolati (variabili, blocchi allocati dinamicamente, ecc.)
- Lo **stato della CPU** (registri interni, ecc.)
- Un insieme di **risorse in uso** (file aperti, ecc.)
- Un insieme di **metadati** che tengono traccia vari aspetti legati al processo stesso e all'esecuzione del programma (identificatore del processo, utente proprietario del processo, per quanto tempo il processo è stato in esecuzione, ecc.)

Un processo può essere attivato in vari modi:

- su **richiesta esplicita dell'utente** che richiede l'esecuzione di un programma: questo può avvenire sotto forma di comandi impartiti da **riga di comando** (si veda l'Appendice C), oppure via **interfaccia grafica** facendo clic sull'icona associata a un programma eseguibile.
- su **richiesta di altri processi**
- **in risposta ad eventi** come lo scadere di un timer usato per attività programmate nel tempo (es. aggiornamento periodico della posta elettronica).

Linux/MacOS X

Per elencare tutti i processi correntemente attivi nel sistema è possibile usare il comando `ps -e`.

```
$ ps -e
```

[...]

3.2 Immagine di memoria di un processo

[...]

3.2.1 Il file system virtuale /proc in Linux

Linux

[...]

```
$ [...]
```

```
[...]
```

4 Come viene tradotto un programma C in linguaggio macchina?

I sistemi di calcolo si basano su un certo numero di **astrazioni** che forniscono una visione più semplice del funzionamento della macchina, nascondendo dettagli dell'implementazione che possono essere, almeno in prima battuta, ignorati.

Due delle più importanti astrazioni sono:

- La **memoria**, vista come un grosso array di byte.
- L'**instruction set architecture** (ISA), che definisce:
    a. lo stato della CPU;
    b. il formato delle sue istruzioni;
    c. l'effetto che le istruzioni hanno sullo stato.

Per tradurre codice di alto livello (ad esempio in linguaggio C) in codice macchina, i compilatori si basano sulla descrizione astratta della macchina data dalla sua ISA.

Due delle ISA più diffuse sono:

- IA32, che descrive le architetture della famiglia di processori x86 a 32 bit;
- x86-64, che descrive le architetture della famiglia di processori x86 a 64 bit.

L'x86-64 è ottenuto come estensione dell'IA32, con cui è **retrocompatibile**. Le istruzioni IA32 sono infatti presenti anche nell'x86-64, ma l'x86-64 introduce **nuove istruzioni** non supportate dal'IA32. Programmi scritti in linguaggio macchina per piattaforme IA32 possono essere eseguiti anche su piattaforme x86-64, ma in generale non vale il viceversa. In questa dispensa tratteremo l'IA32.

4.1 Instruction set architecture (ISA) IA32

4.1.1 Tipi di dato macchina

L'IA32 ha sei tipi di dato numerici primitivi (tipi di dato macchina):

Tipo di dato macchina	Rappresen-tazione	Suffisso assembly	Dimensione in byte
Byte	intero	`b`	1
Word	intero	`w`	2
Double word	intero	`l`	4
Single precision	virgola mobile	`s`	4

Double precision	virgola mobile	`l`	8
Extended precision	virgola mobile	`t`	12 (10 usati)

I tipi macchina permettono di rappresentare sia **numeri interi** che **numeri in virgola mobile**. Si noti che il tipo Extended precision richiede 12 byte in IA32. Tuttavia, di questi solo 10 byte (80 bit) sono effettivamente usati.

Ogni tipo ha un corrispondente **suffisso assembly** che, come vedremo, viene usato per denotare il **tipo degli operandi di una istruzione**.

4.1.2 Corrispondenza tra tipi di dato C e tipi di dato macchina

La seguente tabella mostra la corrispondenza tra i tipi di dato primitivi C (interi, numeri in virgola mobile e puntatori) e i tipi di dato primitivi macchina:

Tipo di dato C	Tipo di dato macchina	Suffisso	Dimensione in byte
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`long`	Double word	`l`	4
`long long`	*non supportato*	-	8
puntatore	Double word	`l`	4
`float`	Single precision	`s`	4
`double`	Double precision	`l`	8
`long double`	Extended precision	`t`	12 (10 usati)

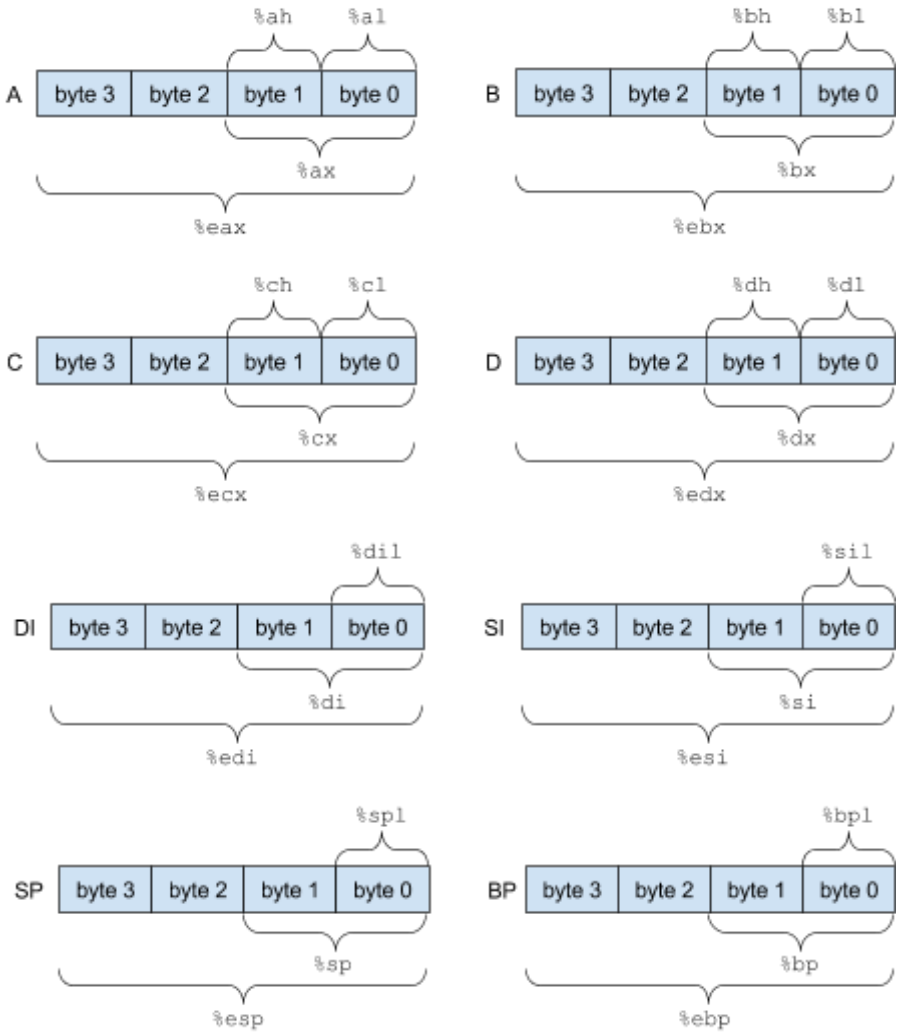
Si noti che le uniche differenze fra IA32 e x86-64 sono sui tipi `long`, puntatore e `long double`, evidenziati nella tabella. Inoltre, il tipo di dato `long long` non è supportato in modo nativo dall'hardware IA32.

Interi con e senza segno hanno il **medesimo tipo macchina** corrispondente: ad esempio, sia `char` che `unsigned char` sono rappresentati come Byte.

4.1.3 Registri

I **registri** sono delle memorie ad altissima velocità a bordo della CPU. In linguaggio assembly, sono identificati mediante dei **nomi simbolici** e possono essere usati in un programma come se fossero variabili.

L'IA32 ha 8 registri interi (A, B, C, D, DI, SI, SP, BP) di dimensione 32 bit (4 byte), di cui i primi 6 possono essere usati come se fossero variabili per memorizzare interi e puntatori:



I registri SP e BP hanno invece un uso particolare che vedremo in seguito. Nella descrizione, $byte_0$ denota il byte **meno significativo** del registro e $byte_3$ quello **più significativo**.

Si noti che è possibile accedere a singole parti di un registro utilizzando dei nomi simbolici. Ad esempio, per il registro A:

- `%eax` denota i 4 byte di A ($byte_3$, $byte_2$, $byte_1$, $byte_0$)
- `%ax` denota i due byte meno significativi di A ($byte_1$ e $byte_0$)
- `%al` denota il byte meno significativo di A ($byte_0$)

- `%ah` denota il secondo byte meno significativo di A ($byte_1$)

4.1.4 Operandi e modi di indirizzamento della memoria

Le istruzioni macchina hanno in genere uno o più **operandi** che definiscono i dati su cui operano. In generale, si ha un **operando sorgente** che specifica un valore di ingresso per l'operazione e un **operando destinazione** che identifica dove deve essere immagazzinato il risultato dell'operazione.

Gli operandi sorgente possono essere di tre tipi:
- *Immediato*: operando immagazzinato insieme all'istruzione stessa;
- *Registro*: operando memorizzato in uno degli 8 registri interi;
- *Memoria*: operando memorizzato in memoria.

Gli operandi destinazione possono essere invece di soli due tipi:
- *Registro*: il risultato dell'operazione viene memorizzato in uno degli 8 registri interi;
- *Memoria*: il risultato dell'operazione viene memorizzato in memoria.

Useremo la seguente notazione:
- Se `E` è il nome di un registro, `R[E]` denota il contenuto del registro `E`;
- Se `x` è un indirizzo di memoria, M_b`[x]` denota dell'oggetto di `b` byte all'indirizzo `x` (omettiamo il pedice `b` quando la dimensione è irrilevante ai fini della descrizione).

Si hanno le seguenti 11 possibili forme di operandi. Per gli operandi di tipo memoria, vi sono vari **modi di indirizzamento** che consentono di accedere alla memoria dopo averne calcolato un indirizzo.

Tipo	Sintassi	Valore denotato	Nome convenzionale
Immediato	`$imm`	`imm`	Immediato
Registro	`E`	`R[E]`	Registro
Memoria	`imm`	`M[imm]`	Assoluto
Memoria	(E_{base})	`M[R[`E_{base}`]]`	Indiretto
Memoria	`imm`(E_{base})	`M[imm+R[`E_{base}`]]`	Base e spiazzamento
Memoria	(E_{base},E_{indice})	`M[R[`E_{base}`]+R[`E_{indice}`]]`	Base e indice
Memoria	`imm`(E_{base},E_{indice})	`M[imm+R[`E_{base}`]+R[`E_{indice}`]]`	Base, indice e spiazzamento
Memoria	(E_{indice},s)	`M[R[`E_{indice}`]·s]`	Indice e scala

Memoria	`imm(E`$_{indice}$`,s)`	`M[imm+R[E`$_{indice}$`]·s]`	Indice, scala e spiazzamento
Memoria	`(E`$_{base}$`,E`$_{indice}$`,s)`	`M[R[E`$_{base}$`]+R[E`$_{indice}$`]·s]`	Base, indice e scala
Memoria	`imm(E`$_{base}$`,E`$_{indice}$`,s)`	`M[imm+R[E`$_{base}$`]+R[E`$_{indice}$`]·s]`	Base, indice, scala e spiazzamento

Negli indirizzamenti a memoria con indice scalato, il parametro `s` può assumere solo uno dei valori: 1, 2, 4, 8. Il parametro immediato imm è un valore intero costante a 32 bit, ad esempio -24 (decimale) oppure 0xAF25CB7E (esadecimale).

Nel seguito, usiamo la notazione S_n per denotare un operando **sorgente** di n byte, e D_n per denotare un operando **destinazione** di n byte. Omettiamo il pedice quando la dimensione è irrilevante ai fini della descrizione.

4.1.5 Rappresentazione dei numeri in memoria: big-endian vs. little-endian

L'**endianess** di un processore definisce l'**ordine** con cui vengono disposti in **memoria** i **byte** della rappresentazione di un valore numerico:

- **big-endian**: il byte **più** significativo del numero viene posto all'indirizzo più basso;
- **little-endian**: il byte **meno** significativo del numero viene posto all'indirizzo più basso.

Ad esempio, l'intero a 32 bit 0x12345678 viene disposto all'indirizzo 0x100 di memoria con le seguenti sequenze di byte (in esadecimale):

Si noti come nel formato big-endian l'ordine dei byte è lo stesso in cui appare nel letterale numerico che denota il numero, in cui la cifra più significativa appare per prima. Nel little-endian è il contrario.

Esempi di processori big endian sono PowerPC e SPARC. Processori little-endian sono ad esempio quelli della famiglia x86.

4.1.6 Istruzioni di movimento dati

Le istruzioni di movimento dati servono per **copiare byte** da memoria a registro, da registro a registro, e da registro a memoria.

4.1.6.1 Stessa dimensione sorgente e destinazione: MOV

Una delle istruzioni più comuni è la `MOV`, dove sorgente e destinazione hanno la stessa dimensione.

Istruzione	Effetto	Descrizione
`MOV S,D`	D ← S	*copia byte da sorgente S a destinazione D*
`movb` S_1`,`D_1	$D_1 \leftarrow S_1$	copia 1 byte
`movw` S_2`,`D_2	$D_2 \leftarrow S_2$	copia 2 byte
`movl` S_4`,`D_4	$D_4 \leftarrow S_4$	copia 4 byte

4.1.6.2 Dimensione destinazione maggiore di quella sorgente: MOVZ, MOVS

Le istruzioni `MOVZ`, e `MOVS` servono per spostare dati da un operando sorgente a un operando destinazione di dimensione maggiore. Servono per effettuare le conversioni di tipi interi senza segno (`MOVZ`) e con segno (`MOVS`).

Istruzione	Effetto	Descrizione
`MOVZ S,D`	D ← ZeroExtend(S)	*copia byte da sorgente S a destinazione D, riempiendo con zero i byte che D ha in più rispetto a S*
`movzbw` S_1`,`D_2	$D_2 \leftarrow 0x00:S_1$	copia 1 byte in 2 byte, estendi con zero
`movzbl` S_1`,`D_4	$D_4 \leftarrow 0x000000:S_1$	copia 1 byte in 4 byte, estendi con zero
`movzwl` S_2`,`D_4	$D_4 \leftarrow 0x0000:S_2$	copia 2 byte in 4 byte, estendi con zero

Esempi:

Si assuma `%eax=0x12341234` e `%ecx=0xABCDEFAB`:

Istruzione	**Risultato** (estensione sottolineata)
`movzbw %al,%cx`	`%eax=0x123412`**`34`** `%ecx=0xABCD`**`00`**`34`
`movzbl %al,%ecx`	`%eax=0x123412`**`34`** `%ecx=0x`**`00000034`**
`movzwl %ax,%ecx`	`%eax=0x1234`**`1234`** `%ecx=0x`**`00001234`**

Vediamo ora l'istruzione `MOVS`:

Istruzione	Effetto	Descrizione
MOVS S,D	D ← SignExtend(S)	*copia byte da sorgente S a destinazione D, riempiendo con il bit del segno (bit più significativo) di S i byte che D ha in più rispetto a S*
movzbw S_1,D_2	D_2 ← 0xMM:S_1	copia 1 byte in 2 byte, estendi con M='F' se il bit più significativo di S_1 (bit del segno) è 1 e con M='0' altrimenti
movzbl S_1,D_4	D_4 ← 0xMMMMMM:S_1	copia 1 byte in 4 byte, estendi con M='F' se il bit più significativo di S_1 (bit del segno) è 1 e con M='0' altrimenti
movzwl S_2,D_4	D_4 ← 0xMMMM:S_2	copia 2 byte in 4 byte, estendi con M='F' se il bit più significativo di S_2 (bit del segno) è 1 e con M='0' altrimenti

Esempi:

Si assuma `%eax=0x12341234` e `%ecx=0xABCDE1E2`:

Istruzione	**Risultato** (estensione sottolineata)
`movsbw %al,%cx`	`%eax=0x12341234` `%ecx=0xABCD0034`
`movsbl %al,%ecx`	`%eax=0x12341234` `%ecx=0x00000034`
`movswl %ax,%ecx`	`%eax=0x12341234` `%ecx=0x00001234`
`movsbw %cl,%ax`	`%ecx=0xABCDE1E2` `%eax=0x1234FFE2`
`movsbl %cl,%eax`	`%ecx=0xABCDE1E2` `%eax=0xFFFFFFE2`
`movswl %cx,%eax`	`%ecx=0xABCDE1E2` `%eax=0xFFFFE1E2`

4.1.6.3 Movimento dati da/verso la stack: PUSH, POP

Le istruzioni `PUSH`, e `POP` servono per spostare dati da un operando sorgente verso la cima della stack (`PUSH`) e dalla cima della stack verso un operando destinazione (`POP`):

Istruzione	Effetto	Descrizione
pushl S_4	R[%esp] ← R[%esp] - 4	copia l'operando di 4 byte S sulla cima della

	`M[R[%esp]] ← S4`	stack
`popl D4`	`D4 ← M[R[%esp]]` `R[%esp] ← R[%esp] + 4`	togli i 4 byte dalla cima della stack e copiali nell'operando D

4.1.7 Istruzioni aritmetico-logiche

Le seguenti istruzioni IA32 servono per effettuare operazioni su interi a 1, 2 e 4 byte:

Istruzione	Effetto	Descrizione
`INC D`	D ← D+1	incrementa destinazione
`DEC D`	D ← D-1	decrementa destinazione
`NEG D`	D ← -D	inverti segno destinazione
`NOT D`	D ← ~D	complementa a 1 destinazione
`ADD S,D`	D ← D+S	aggiungi sorgente a destinazione e risultato in destinazione
`SUM S,D`	D ← D-S	sottrai sorgente da destinazione e risultato in destinazione
`IMUL S,D`	D ← D*S	moltiplica sorgente con destinazione e risultato in destinazione
`XOR S,D`	D ← D^S	or esclusivo sorgente con destinazione e risultato in destinazione
`OR S,D`	D ← D|S	or sorgente con destinazione e risultato in destinazione
`AND S,D`	D ← D&S	and sorgente con destinazione e risultato in destinazione

Omettiamo per il momento istruzioni più complesse come quelle che effettuano divisioni.

4.1.7.1 L'istruzione LEA (load effective address)

L'istruzione LEA consente di sfruttare la flessibilità data dai modi di indirizzamento a memoria per calcolare espressioni aritmetiche che coinvolgono somme e prodotti su indirizzi o interi.

Istruzione	Effetto	Descrizione
`leal S,D4`	`D4 ← &S`	Calcola l'indirizzo effettivo specificato dall'operando di tipo memoria S e lo scrive in D

Si noti che `leal`, diversamente da `movl`, non effettua un accesso a memoria sull'operando sorgente. L'istruzione `leal` calcola infatti l'indirizzo effettivo dell'operando sorgente, senza però accedere in memoria a quell'indirizzo.

Esempi.

Si assuma `%eax=0x100`, `%ecx=0x7` e M_4`[0x100]=0xCAFE`

Istruzione	Effetto	Risultato
`movl (%eax),%edx`	`R[%edx]` ← M_4`[R[%eax]]`	`%edx=0xCAFE`
`leal (%eax),%edx`	`R[%edx]` ← `R[%eax]`	`%edx=0x100`

Si noti la differenza fra `leal` e `movl` che abbiamo discusso sopra. Si considerino inoltre i seguenti altri esempi:

Istruzione	Effetto	Risultato
`leal (%eax,%ecx),%edx`	`R[%edx]` ← `R[%eax]+R[%ecx]`	`%edx=0x107`
`leal -3(%eax,%ecx),%edx`	`R[%edx]` ← `R[%eax]+R[%ecx]-3`	`%edx=0x104`
`leal -3(%eax,%ecx,2),%edx`	`R[%edx]` ← `R[%eax]+R[%ecx]·2-3`	`%edx=0x10B`
`leal (%eax,%ecx,2),%edx`	`R[%edx]` ← `R[%eax]+R[%ecx]·2`	`%edx=0x10E`
`leal (%ecx,4),%edx`	`R[%edx]` ← `R[%ecx]·4`	`%edx=0x1C`

L'istruzione `leal` viene usata per scrivere **programmi più veloci** e viene sfruttata tipicamente per due scopi:

1. calcolare l'indirizzo effettivo di un oggetto in memoria una sola volta, per poi usarlo più volte;
2. calcolare **espressioni aritmetiche su interi o puntatori** usando una sola istruzione.

Si noti infatti che, sebbene sia stata pensata per calcolare indirizzi di memoria, la `leal` può essere usata per calcolare espressioni intere che non rappresentano indirizzi.

Esempio.

Si consideri il seguente frammento di programma C:

```
int x=10;
int y=20;
int z=x+y*4-7;
```

Riformuliamo il frammento in modo che ogni operazione aritmetica abbia la forma: `a = a op b`, ottenendo il seguente codice equivalente, la corrispondente traduzione in codice IA32 e una versione ottimizzata del codice IA32 basata sull'istruzione `leal`:

Codice C	Codice IA32	Codice IA32 ottimizzato

`int x=10; // x è in %eax`
`int y=20; // y è in %ecx`
`int z=y;  // z è in %edx`
`z=z*4;`
`z=z-7;`
`z=z+x;`
	`movl $10,%eax`
	`movl $20,%ecx`
	`movl %ecx,%edx`
	`imull $4,%edx`
	`addl $-7,%edx`
	`addl %eax,%edx`
		`movl $10,%eax`
		`movl $20,%ecx`
		`leal -7(%eax,%ecx,4),%edx`

Si noti che, se l'espressione da calcolare fosse stata `x+y*5-7`, non sarebbe stato possibile usare la `leal`: infatti il fattore moltiplicativo nei vari modi di indirizzamento a memoria (scala) può essere solo 1, 2, 4, 8. Non tutte le espressioni aritmetiche possono quindi essere calcolate con la `leal`.

4.1.8 Istruzioni di salto

Normalmente, il flusso del controllo di un programma procede in modo sequenziale, eseguendo le istruzioni nell'ordine in cui appaiono in memoria. Ogni volta che un'istruzione I viene eseguita, il registro EIP (instruction pointer), che punta alla prossima istruzione da eseguire, viene incrementato automaticamente del numero di byte occupati dall'istruzione I.

Vi sono tuttavia istruzioni, chiamate **istruzioni di salto**, che permettono di alterare il flusso del controllo, modificando il contenuto del registro EIP in modo che l'esecuzione non prosegua con istruzione successiva, ma con un'altra che inizia ad un indirizzo diverso.

Vi sono tre tipi di istruzioni di salto:

1. salti **incondizionati**: il registro EIP viene sovrascritto con l'indirizzo di memoria dell'istruzione a cui si vuole saltare;
2. salti **condizionati**: il registro EIP viene sovrascritto con l'indirizzo di memoria dell'istruzione a cui si vuole saltare, ma solo se è verificata una determinata condizione sui dati;
3. **chiamata** e **ritorno** da funzione (che vedremo in seguito).

[...]

Appendice A: tabella dei caratteri ASCII a 7 bit

Le seguenti tabelle contengono i 127 caratteri della codifica ASCII base (7 bit), divisi in caratteri di controllo (da 0 a 31) e caratteri stampabili (da 32 a 126). Nella tabella riportiamo i codici numerici associati ai caratteri in varie basi (DEC=10, OCT=8, HEX=16).

A.1 Caratteri ASCII di controllo

I primi 32 caratteri sono caratteri non stampabili utilizzati storicamente per controllare periferiche come stampanti. Fra di essi, ci sono i codici che rappresentano i ritorni a capo (caratteri 10 e 13) e il carattere di tabulazione (carattere 9). Nella colonna C riportiamo i codici di escape usati nel linguaggio C per rappresentare alcuni dei caratteri di controllo nelle stringhe e nei letterali `char`.

DEC	OCT	HEX	Simbolo	C	Descrizione
0	000	00	NUL	\0	Null char
1	001	01	SOH		Start of Heading
2	002	02	STX		Start of Text
3	003	03	ETX		End of Text
4	004	04	EOT		End of Transmission
5	005	05	ENQ		Enquiry
6	006	06	ACK		Acknowledgment
7	007	07	BEL	\a	Bell
8	010	08	BS	\b	Back Space
9	011	09	HT	\t	Horizontal Tab
10	012	0A	LF	\n	Line Feed
11	013	0B	VT	\v	Vertical Tab
12	014	0C	FF	\f	Form Feed
13	015	0D	CR	\r	Carriage Return

14	016	0E	SO		Shift Out / X-On
15	017	0F	SI		Shift In / X-Off
16	020	10	DLE		Data Line Escape
17	021	11	DC1		Device Control 1 (oft. XON)
18	022	12	DC2		Device Control 2
19	023	13	DC3		Device Control 3 (oft. XOFF)
20	024	14	DC4		Device Control 4
21	025	15	NAK		Negative Acknowledgement
22	026	16	SYN		Synchronous Idle
23	027	17	ETB		End of Transmit Block
24	030	18	CAN		Cancel
25	031	19	EM		End of Medium
26	032	1A	SUB		Substitute
27	033	1B	ESC	\e	Escape
28	034	1C	FS		File Separator
29	035	1D	GS		Group Separator
30	036	1E	RS		Record Separator
31	037	1F	US		Unit Separator

A.2 Caratteri ASCII stampabili

Vi sono 95 caratteri stampabili, con codici compresi tra 32 e 126:

DEC	OCT	HEX	Simbolo
32	040	20	*spazio*
33	041	21	!

DEC	OCT	HEX	Simbolo
80	120	50	P
81	121	51	Q

34	042	22	"
35	043	23	#
36	044	24	$
37	045	25	%
38	046	26	&
39	047	27	'
40	050	28	(
41	051	29	)
42	052	2A	*
43	053	2B	+
44	054	2C	,
45	055	2D	-
46	056	2E	.
47	057	2F	/
48	060	30	0
49	061	31	1
50	062	32	2
51	063	33	3
52	064	34	4
53	065	35	5
54	066	36	6
55	067	37	7
56	070	38	8
57	071	39	9
58	072	3A	:

82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	\
93	135	5D	]
94	136	5E	^
95	137	5F	_
96	140	60	`
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j

59	073	3B	;
60	074	3C	<
61	075	3D	=
62	076	3E	>
63	077	3F	?
64	100	40	@
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O

107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	|
125	175	7D	}
126	176	7E	~

Appendice B: il file system

Il **file system** denota l'insieme di funzionalità e schemi organizzativi con cui un sistema di calcolo si interfaccia con i dispositivi di archiviazione, consentendone la gestisce dei dati.

B.1.1 File e directory

Un **file** (archivio) è una sorgente (o un deposito) di informazioni accessibili in lettura e/o in scrittura a un programma. Normalmente è costituito da una sequenza di byte memorizzati in forma permanente su disco.

I file sono organizzati in **directory**, che sono dei contenitori che possono contenere file e altre directory.

File e directory hanno un **nome** che li identifica e altre **proprietà**, fra cui un elenco di **permessi** che specificano quali utenti abbiano diritto di leggerli/scriverli.

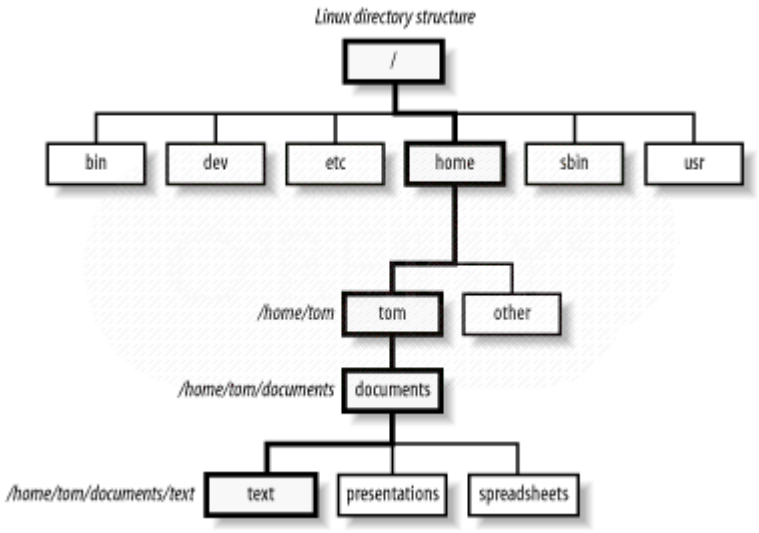
I nomi dei file hanno spesso delle **estensioni** della forma .estensione, che descrivono la natura del loro contenuto. Ad esempio, l'estensione .c denota file di testo scritti in C (es. hello.c), l'estensione .jpg indica file binari che contengono immagini (es. foto-montagna.jpg), ecc.

B.1.1 L'albero delle directory

La struttura delle directory è tipicamente ad albero: la **directory radice**, denotata da /, contiene tutti i dati memorizzati in forma permanente nel sistema sotto forma di file e sottodirectory.

In questo corso useremo come caso di studio i sistemi UNIX e UNIX-like (Linux, MacOS X, ecc.), denotati con l'abbreviazione *nix.

L'organizzazione tipica delle directory di un sistema *nix, chiamata Filesystem Hierarchy Standard (FHS), è la seguente (semplificata):



Ogni directory, tranne la directory radice ha una **directory genitore**. Se una directory è contenuta in un'altra, allora è una **directory figlia**. Nell'esempio sopra, `tom` e `other` sono figlie di `home`, mentre `home` è genitore di `tom` e `other`.

B.1.2 Percorso assoluto e percorso relativo

Ogni file o directory è identificato da un **percorso** (path), che ne identifica la posizione nella struttura delle directory. Un percorso può essere assoluto o relativo.

Percorso assoluto

Elenca tutte le directory che bisogna attraversare per arrivare al punto desiderato a partire dalla directory radice `/`.

Esempio: `/home/anna/Scrivania/foto-montagna.jpg` è il percorso assoluto di un file chiamato `foto-montagna.jpg` posizionato nella directory `Scrivania` localizzata nella `home` directory dell'utente `anna`.

Percorso relativo

Descrive la posizione relativa di una directory rispetto ad un'altra presa come riferimento.

Esempio: il percorso relativo del file `/home/anna/Scrivania/foto-montagna.jpg` rispetto alla directory `/home` è: `anna/Scrivania/foto-montagna.jpg`.

Ci sono due percorsi relativi particolari:

- `..` (**doppio punto**): è un percorso relativo che denota la **directory genitore**. Ad esempio,

il percorso relativo `../Documenti` può essere letto come: sali alla directory genitore e poi entra nella directory figlia Documenti. Relativamente a `/home/anna/Scrivania`, `../Documenti` denota `/home/anna/Documenti`.

- `.` (**punto**): è un percorso relativo che denota la **directory stessa**. Ad esempio, il percorso `./hello` denota il file chiamato `hello` nella directory di riferimento.

Appendice C: la shell dei comandi

Una **shell** è un programma che consente l'immissione in forma testuale di comandi che devono essere eseguiti dal sistema di calcolo, realizzando quella che viene chiamata **interfaccia a riga di comando** (in inglese: command-line interface, o CLI). In questa dispensa usiamo come shell il programma bash, usato come shell di default in MacOS X e nelle maggiori distribuzioni Linux.

Aprendo una finestra di terminale, si attiva una shell che mostra la **riga di comando**: nella shell bash la riga di comando è normalmente indicata dal simbolo $ seguito dal cursore ▉. Lo scopo del simbolo $ è quello di avvisare l'utente che la shell è **pronta a ricevere comandi**. Il simbolo $ è normalmente preceduto da informazioni sull'utente che sta lavorando, sul nome del computer e sulla directory corrente.

Esempio:

```
studente@c1565:~$ ▉
```

`studente` è il nome dell'utente autenticato, `c1565` è il nome del computer e `~` è la directory corrente (home dell'utente).

Come osservato, in ogni istante la shell è posizionata in una **`directory corrente`**. All'avvio del terminale, la directory corrente è normalmente la directory home `d`ell'utente con cui ci si è autenticati, indicata dal simbolo `~`. La home directory raccoglie tutti i file, le directory e le impostazioni dell'utente.

Ogni **comando** ha la forma: `nome-comando [parametri]`.

Per far **eseguire un comando** alla shell, lo si digita nel terminale e poi si preme il tasto Invio (Return) ⏎. I parametri sono opzionali.

Vi sono due tipi di comandi:

- **Comandi esterni**: `nome-comando` è il nome di un file eseguibile. L'esecuzione del comando lancia un nuovo processo basato sull'eseguibile indicato;

- **Comandi interni (built-in)**: `nome-comando` è un comando eseguito direttamente dalla shell e non provoca l'esecuzione di nuovi processi.

I **percorsi relativi** sono sempre **riferiti** alla **directory corrente** della shell.

Segue un elenco dei comandi interni ed esterni più comunemente usati.

Comando	Tipo	Descrizione
pwd	interno	visualizza il percorso assoluto della directory corrente
cd	interno	cambia directory corrente
ls	esterno	elenca il contenuto di una directory
touch	esterno	crea un file vuoto o ne aggiorna la data di modifica
mv	esterno	rinomina o sposta un file o una directory
mkdir	esterno	crea una nuova directory vuota
rmdir	esterno	elimina una directory, purché sia vuota
rm	esterno	elimina un file o una directory
cp	esterno	copia un file o una directory

C.1 Manipolazione ed esplorazione del file system

C.1.1 pwd: visualizza il percorso assoluto della directory corrente

```
pwd
```

Visualizza il percorso assoluto della directory corrente.

Esempio:

```
$ pwd
/home/studente/Scrivania
```

C.1.2 cd: cambia directory corrente

```
cd nome-directory
```

Usa come directory corrente quella specificata dal percorso (assoluto o relativo) `nome-directory`.

Esempio 1:
`$ cd /home/studente`
posiziona la shell nella directory `/home/studente`.

Esempio 2:
`$ cd ..`
posiziona la shell nella directory genitore di quella corrente.

Esempio 3:
`$ cd ../Scrivania`
posiziona la shell nella directory `Scrivania` contenuta nella genitore di quella corrente.

```
cd
```

Posiziona la shell nella home directory dell'utente corrente.

C.1.3 ls: elenca il contenuto di una directory

```
ls [nome-directory]
```

Elenca il contenuto directory specificata dal percorso (assoluto o relativo) `nome-directory`. Se `nome-directory` è assente, elenca il contenuto della directory corrente.

```
ls -l [nome-directory]
```

Elenca il contenuto della directory corrente, fornendo maggiori informazioni (se è un file o directory, la dimensione del file, la data di modifica e altro). Se `nome-directory` è assente, elenca il contenuto della directory corrente.

Esempio output 1:

```
-rw-r--r--    9 anna  staff      306 Oct  8 18:10 hello.c
```

indica (fra altre cose) che `hello.c` è un file e non una directory (la riga inizia per `-`), può essere letto e scritto dall'utente anna (`rw`) occupa 306 byte, ed è stato modificato l'8 ottobre alle 18:10.

Esempio output 2:

```
drwxr-xr-x    8 studente  staff      272 Sep 27 13:16 foto
```

indica (fra altre cose) che `foto` è una directory (la riga inizia per `d`) può essere letta, scritta e listata dall'utente studente (`rwx`), e il suo contenuto è stato modificato il 27 settembre alle 13:16.

C.1.4 touch: crea un file vuoto o ne aggiorna la data di modifica

```
touch nome-file
```

Crea un file vuoto `nome-file` o ne aggiorna la data di modifica se esiste già.

Esempio 1:
`$ touch hello.c`
crea il file vuoto `hello.c` nella directory corrente o ne aggiorna la data di modifica se esiste già.

Esempio 2:
`$ touch /home/studente/Scrivania/hello.c`
crea il file vuoto `hello.c` sulla scrivania dell'utente `studente`, o ne aggiorna la data di modifica se esiste già.

C.1.5 mv: rinomina o sposta un file o una directory

```
mv sorgente destinazione
```

Rinomina il file o la directory sorgente con il nuovo nome destinazione, **purché non esista già una directory con il nome destinazione**.

Esempio 1:
`$ mv hello.c hello2.c`
rinomina il file `hello.c` nella directory corrente con il nuovo nome `hello2.c`.

Esempio 2:
`$ mv pippo pluto`
rinomina la directory `pippo` con il nuovo nome `pluto`, assumendo che non esista già nella directory corrente una directory chiamata `pluto`.

```
mv sorgente directory
```

Sposta il file o la directory `sorgente` nella directory `directory`.

Esempio 1:
`$ mv hello.c pippo`
sposta il file `hello.c` nella directory `pippo` (assumendo che `hello.c` e `pippo` siano nella directory corrente).

Esempio 2:
`$ mv pluto pippo`
sposta la directory `pluto` nella directory `pippo` (assumendo che `pluto` e `pippo` siano nella directory corrente).

C.1.6 mkdir: crea una nuova directory vuota

```
mkdir directory
```

Crea una nuova directory vuota `directory`.
Esempio 1:
`$ mkdir pippo`
crea la directory `pippo` nella directory corrente.

Esempio 2:
`$ mkdir /home/studente/Scrivania/pippo`
crea la directory `pippo` nella directory `/home/studente/Scrivania`.

C.1.7 rmdir: elimina una directory, purché sia vuota

```
rmdir directory
```

Elimina la directory `directory`, purché sia vuota.

Esempio 1:
`rmdir pippo`
elimina la directory pippo dalla directory corrente.

Esempio 2:
`rmdir /home/studente/Scrivania/pippo`
elimina la directory `pippo` dalla directory `/home/studente/Scrivania`.

Nota: per eliminare directory non vuote, si veda il comando `rm`.

C.1.8 rm: elimina un file o una directory

```
rm file
```

Elimina il file `file`.

Esempio 1:
```
$ rm hello.c
```
elimina il file `hello.c` dalla directory corrente.

Esempio 2:
```
$ rm /home/studente/Scrivania/hello.c
```
elimina il file `hello.c` dalla directory `/home/studente/Scrivania`.

```
rm -rf directory
```

Elimina la directory `directory` e tutto il suo contenuto di file e sottodirectory.

Esempio 1:
```
$ rm -rf pippo
```
elimina la directory `pippo` e tutto il suo contenuto dalla directory corrente.

Esempio 2:
```
$ rm -rf /home/studente/Scrivania/pippo
```
elimina le directory `pippo` e tutto il suo contenuto dalla directory `/home/studente/Scrivania`.

C.1.9 cp: copia un file o una directory

```
cp file nuovo-file
```

Copia il file `file` creandone uno nuovo chiamato `nuovo-file`.

Esempio 1:
```
$ cp hello.c hello-copia.c
```
copia il file `hello.c` creandone uno nuovo chiamato `hello-copia.c`.

Esempio 2:
`$ cp /home/studente/Scrivania/hello.c  ../hello-copia.c`
copia il file `hello.c` dalla directory `/home/studente/Scrivania` nella directory genitore di quella corrente con il nome `hello-copia.c`.

`cp -R directory nuova-directory`

Copia la directory `directory` e tutto il suo contenuto di file e sottocartelle creandone una nuova dal nome `nuova-directory`.

Esempio 1:
`$ cp -R pippo pluto`
copia la directory `pippo` e tutto il suo contenuto di file e sottocartelle creandone una nuova chiamata `pluto`.

Esempio 2:
`$ cp -R /home/studente/Scrivania/pippo /home/studente/Scrivania/pluto`
copia la directory `pippo` e tutto il suo contenuto creandone una nuova chiamata `pluto` nella directory `/home/studente/Scrivania`.

`cp file directory-esistente`

Copia il file `file` nella directory `directory-esistente`.

Esempio 1:
`$ cp hello.c pippo`
copia il file `hello.c` creandone una copia nella directory (esistente) `pippo`.

Esempio 2:
`$ cp /home/studente/Scrivania/hello.c .`
copia il file `hello.c` dalla directory `/home/studente/Scrivania` nella directory corrente.

`cp -R directory directory-esistente`

Copia la directory `directory` e tutto il suo contenuto di file e sottodirectory nella directory esistente `directory-esistente`.

Esempio 1:
`$ cp -R pippo pluto`
copia la directory `pippo` e tutto il suo contenuto di file e sottodirectory nella directory esistente

`pluto`.

Esempio 2:
`$ cp -R /home/studente/Scrivania/pippo .`
copia la directory `pippo` e tutto il suo contenuto dalla directory `/home/studente/Scrivania` nella directory corrente.

`cp -R directory/ directory-esistente`

Copia il **contenuto** della directory `directory` inclusi file e sottodirectory nella directory esistente `directory-esistente`.
 	
Esempio 1:
`$ cp -R pippo/ pluto`
copia il contenuto della directory `pippo` inclusi file e sottodirectory nella directory esistente `pluto`.
 	
Esempio 2:
`$ cp -R /home/studente/Scrivania/pippo/ .`
copia il contenuto della directory `/home/studente/Scrivania/pippo` inclusi file e sottodirectory nella directory corrente.

C.2 Altri comandi utili

geany	Lancia l'editor di testo `geany`. `geany &`: apre l'editor di testo `geany`. `geany file &`: apre il file di testo `file` nell'editor di testo `geany`. *Esempio:* `geany hello.c &` apre il file `hello.c` usando l'editor di testo `geany`.
cat	Invia il contenuto di un file sul canale di output standard (di default è il terminale). `cat file`: invia il contenuto del file file sul canale di output standard (di default è il terminale).
less	Visualizza nel terminale il contenuto di un file, consentendone di scorrere il contenuto con le frecce.

	`less file`: visualizza il contenuto del file file nel terminale (premere `q` per uscire).

man	Visualizza nel terminale la documentazione di un comando esterno.
	
	`man comando-esterno`: visualizza nel terminale la documentazione del comando esterno comando-esterno.
	
	Esempio 1:
	`man cp`
	visualizza nel terminale la documentazione del comando cp.
	
	Esempio 2:
	`man man`
	visualizza nel terminale la documentazione del comando man.