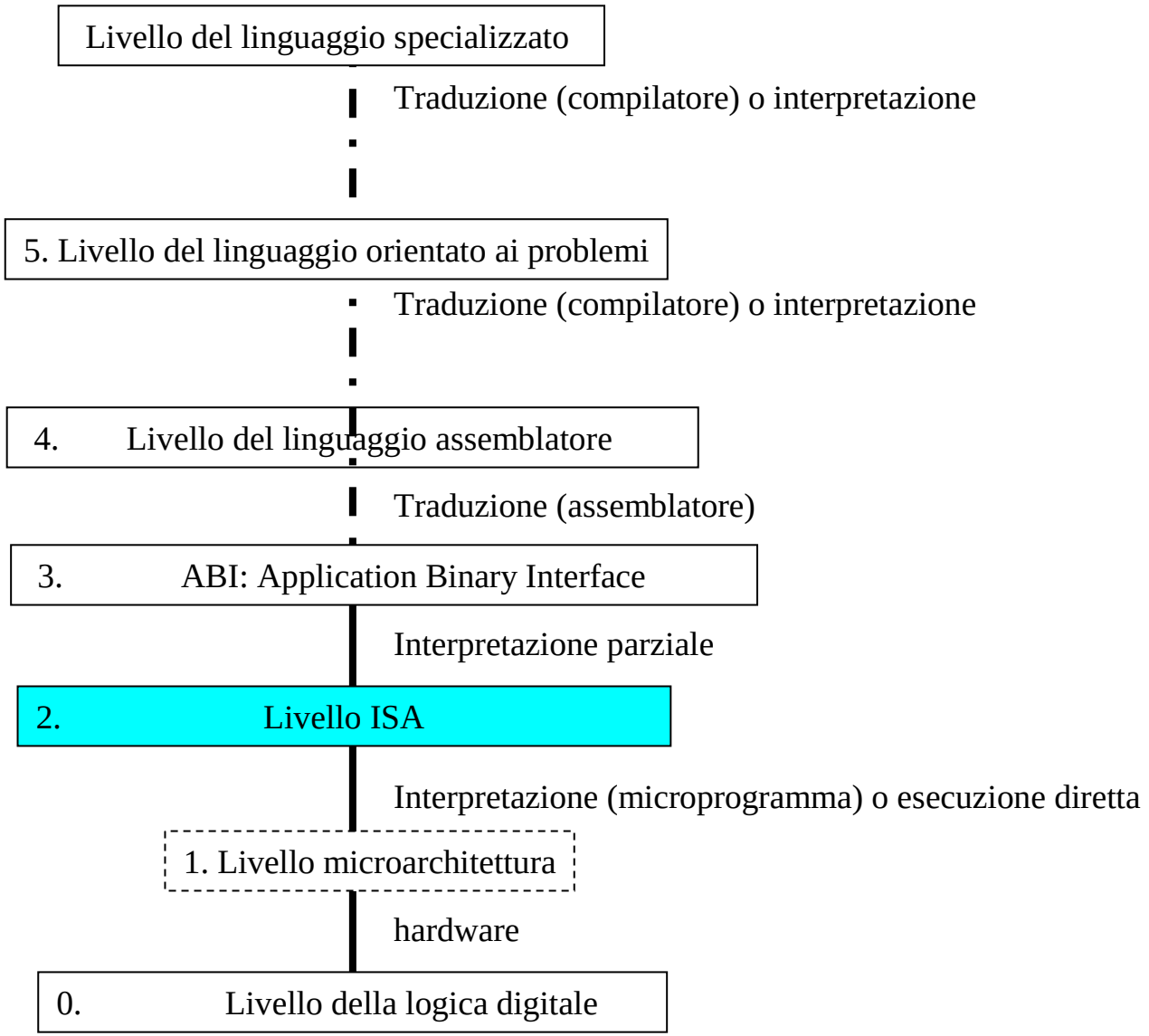
Calcolatori Elettronici A
a.a. 2008/2009

Instruction Set Architecture: nozioni generali

Massimiliano Giacomin

DOVE CI TROVIAMO…



Oggi il linguaggio assembly è sempre meno usato

- E’ specifico per una particolare macchina (o, meglio architettura ISA): se questa cambia, è necessario riscrivere completamente il codice
- Lunghezza dei programmi maggiore rispetto a linguaggi di alto livello
  $\Rightarrow$ diminuzione produttività dei programmatori
- Mancanza di astrazione: bassa leggibilità del codice, aumento degli errori
- Miglioramento dei compilatori, che sono in grado di ottimizzare il codice anche tenendo conto delle specificità dell’hardware meglio rispetto ad un programmatore (cfr. pipeline)

Oggi il linguaggio assembly è sempre meno usato

- E’ specifico per una particolare macchina (o, meglio architettura ISA): se questa cambia, è necessario riscrivere completamente il codice
- Lunghezza dei programmi maggiore rispetto a linguaggi di alto livello
  $\Rightarrow$ diminuzione produttività dei programmatori
- Mancanza di astrazione: bassa leggibilità del codice, aumento degli errori
- Miglioramento dei compilatori, che sono in grado di ottimizzare il codice anche tenendo conto delle specificità dell’hardware meglio rispetto ad un programmatore (cfr. pipeline)

E allora, perché studiare una ISA?

Oggi il linguaggio assembly è sempre meno usato

- E' specifico per una particolare macchina (o, meglio architettura ISA): se questa cambia, è necessario riscrivere completamente il codice
- Lunghezza dei programmi maggiore rispetto a linguaggi di alto livello
  $\Rightarrow$ diminuzione produttività dei programmatori
- Mancanza di astrazione: bassa leggibilità del codice, aumento degli errori
- Miglioramento dei compilatori, che sono in grado di ottimizzare il codice anche tenendo conto delle specificità dell'hardware meglio rispetto ad un programmatore (cfr. pipeline)

E allora, perché studiare una ISA?

- Perché è importante sapere di cosa si tratta (conoscere ciò con cui si ha a che fare)!
- Perché per capire come è costruito un processore dobbiamo partire dalle sue "specifiche"
- Perché talvolta è usato:
  - nei sistemi embedded: riduzione lunghezza del codice (costo) + vincoli real time
  - in un approccio ibrido, per codificare parti del programma "time critical" (es: superamento delle convenzioni nella codifica di procedure)
  - per utilizzare istruzioni specializzate che un compilatore potrebbe non sfruttare
  - in alcuni calcolatori non è disponibile un linguaggio di alto livello!

TIPOLOGIE DI INSTRUCTION SET ARCHITECTURES

- Sulla base di “dove sono localizzati” gli operandi delle istruzioni
  - architettura a “registro accumulatore” (accumulator architecture)
  - architettura a registri specializzati (special-purpose register architecture)
  - architettura basata su stack (stack architecture)
  - architettura a registri di uso generale (general purpose register architecture)
- Istruzioni a lunghezza fissa vs. variabile
- CISC (Complex Instruction Set Computer)
  vs. RISC (Reduced Instruction Set Computer)

Architettura a “registro accumulatore”

- E’ la prima architettura storicamente sviluppata
- Nei primi calcolatori l’hardware era una risorsa molto costosa:

  $\Rightarrow$ un solo registro, chiamato accumulatore!
- Nelle istruzioni aritmetico/logiche, in generale si hanno:

  due operandi sorgente + un operando destinazione

  $\Rightarrow$ l’accumulatore funge sia da sorgente sia da destinazione;

  $\Rightarrow$ l’altro operando deve essere in memoria!
- Essendo l’unico registro, l’accumulatore non viene indicato (né rappresentato) nelle istruzioni

ESEMPIO: A = B+C (A, B, C in memoria)

Architettura a “registro accumulatore”

- E’ la prima architettura storicamente sviluppata
- Nei primi calcolatori l’hardware era una risorsa molto costosa:

  $\Rightarrow$ un solo registro, chiamato accumulatore!
- Nelle istruzioni aritmetico/logiche, in generale si hanno:

  due operandi sorgente + un operando destinazione

  $\Rightarrow$ l’accumulatore funge sia da sorgente sia da destinazione;

  $\Rightarrow$ l’altro operando deve essere in memoria!
- Essendo l’unico registro, l’accumulatore non viene indicato (né rappresentato) nelle istruzioni

ESEMPIO: A = B+C (A, B, C in memoria)

```
Load B          # acc = B
Add C           # acc = acc+C        (acc = B+C)
Store A         # A = acc            (A   = B+C)
```

PRO

- Risparmio hardware
- Istruzioni macchina più piccole (l'operando accumulatore è sempre implicito)

  $\Rightarrow$ risparmio memoria

CONTRO

- Richiede molti accessi in memoria

  (spilling dei registri: mettere i valori meno usati in memoria)

Architettura a “registri specializzati”

- Evoluzione della architettura a registro accumulatore: vengono aggiunti ***registri specializzati***
- Esempi:
  - registri base e indice per l’accesso in memoria
  - registri che possono contenere dati per le istruzioni aritmetiche
  - registri che puntano all’elemento in cima allo stack, ecc.
- Altri nomi per l’architettura:

  extended accumulator, dedicated register, special-purpose register
- Esempio di processore: Intel 8086

Architettura basata su stack

- Vengono eliminati tutti i registri, sostituiti da uno ***stack***
- Sono previste istruzioni per:
  - ***push*** nello stack di un elemento dalla memoria
  - ***pop*** dallo stack di un elemento in memoria
  - effettuare un'operazione aritmetico-logica sui due elementi in cima allo stack

ESEMPIO: A = B+C (A, B, C in memoria)

```
push B
push C
Add
pop A
```

PRO

- Rendono più semplice il compito dei compilatori
  (che non devono decidere come allocare i registri):
  cfr. uso del *p-code* come linguaggio intermedio
- Istruzioni macchina piccole (nessun operando per le aritmetico-logiche)

CONTRO

- molti accessi in memoria (tutto ciò che non è nello stack è in memoria)

Architettura a registri di uso generale

- I registri possono essere utilizzati da tutte le istruzioni per qualunque scopo
- Si dividono in due categorie:
  - *register-memory* architecture
    (esistono istruzioni aritmetico-logiche con un operando in memoria)
  - *register-register* architecture (detta anche *load-store* architecture)
    (le istruzioni aritmetico-logiche lavorano solo su registri

  Tipicamente, la prima corrisponde ad una evoluzione delle architetture più antiche verso un'architettura a registri di uso generale
- Esistono anche architetture *memory-memory* (es. VAX della DEC)

ESEMPIO: A = B+C (A, B, C in memoria)

```
Load R1, B
Load R2, C
Add R3, R1, R2
Store R3, A
```

PRO

- Minimizzano gli accessi in memoria
  (si possono usare i registri per mantenere molti valori intermedi)

CONTRO

- Istruzioni più lunghe
  (è necessario prevedere appositi campi con cui indicare i registri operando)
- Il lavoro dei compilatori è complesso
  (scelta di quali registri usare per minimizzare lo spilling)

Istruzioni a lunghezza fissa vs. variabile

- Per mantenere la lunghezza del codice quanto più piccola possibile, alcune architetture prevedono una lunghezza delle istruzioni variabile

  Esempi:

  - Intel IA-32: da 1 a 17 bytes
  - IBM 360 (usata nei mainframe): 2, 4, 6 bytes
  - VAX: da 1 a 54 bytes

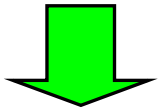
- Per semplificare il compito dell'hardware, altre architetture prevedono una lunghezza costante per tutte le istruzioni macchina

CISC vs. RISC

- In una certa fase dell'evoluzione delle architetture, sviluppo di ISA con istruzioni sempre più complesse (CISC):
  - minimizzazione della lunghezza del codice (risparmio memoria)
  - minimizzazione del numero di istruzioni eseguite, operazioni complesse "cablate" nell'hardware (presunto miglioramento delle prestazioni)
  - semplificazione del compito del programmatore assembly
  - favorite dall'avvento della microprogrammazione
- RISC: la filosofia opposta! (Reduced Instruction Set Computer)

LE TENDENZE ODIERNE

- Densità di integrazione hardware sempre maggiori, costo minore
  $\Rightarrow$ spinta verso le prestazioni, vincoli minori rispetto alla memoria
- Ruolo più limitato della microprogrammazione (fa eccezione il Pentium)
  per l'introduzione della cache interna al processore
- Organizzazione del calcolatore basata su pipeline
- Enorme miglioramento della tecnologia dei compilatori



- Architetture RISC, con formati regolari e lunghezza costante delle istruzioni:
  necessarie per ottenere una semplificazione del compito dell'hardware (pipeline)
  + più efficiente un codice con molte istruzioni elementari (eseguite in parallelo)
  + compilatori hanno un compito semplificato nel contesto di una pipeline
- Adozione di architetture a registri general-purpose o "load-store":
  compilatori sofisticati possono minimizzare gli accessi in memoria
  + lunghezza del codice molto meno importante
  + limitazione dei modi di indirizzamento, sempre per semplificare l'hardware

NUMERO DI REGISTRI GENERAL-PURPOSE

EDSAC	1	accumulator	1949
IBM 701	1	accumulator	1953
CDC 6600	8	load-store	1963
IBM 360	16	register-memory	1964
DEC PDP-8	1	accumulator	1965
DEC PDP-11	8	register-memory	1970
Intel 8008	1	accumulator	1972
Motorola 6800	2	accumulator	1974
DEC VAX	16	register-memory, memory-memory	1977
Intel 8086	1	extended accumulator	1978
Motorola 68000	16	register-memory	1980
Intel 80386	8	register-memory	1985
MIPS	32	load-store	1985
HP PA-RISC	32	load-store	1986
SPARC	32	load-store	1987
PowerPC	32	load-store	1992
DEC Alpha	32	load-store	1992
HP/Intel IA-64	128	load-store	2001
AMD64 (EMT64)	16	register-memory	2003

Una illustre eccezione: Intel IA-32

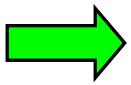
- Ha una lunga storia:
  - Intel 4004, 1972 (accumulator, 4 bit)
  - Intel 8008, 1972 (accumulator, 8 bit)
  - . . .
- Risultato del lavoro indipendente di molti gruppi, aggiunta progressiva al set di istruzioni originali
- Esigenza: mantenere compatibilità all'indietro

⇒
- Istruzioni da 1 a 17 bytes
- Formati delle istruzioni molto complicate
- Filosofia CISC (alcune istruzioni di fatto non sono mai utilizzate)
- Architettura register-memory, con 7 diversi modi di indirizzamento alla memoria

  ⇒ per una add (che può avere operandi entrambi nei registri o uno solo in memoria), vi sono decine di varianti

Una nuova esigenza

- Nei sistemi embedded (es. cellulari) il costo e le dimensioni di memoria ritornano ad essere relativamente importanti
- Se il codice viene scaricato attraverso una rete cellulare, attenzione alla sua lunghezza



Cfr. architetture a stack + istruzioni a lunghezza variabile…?