CALCOLATORI

Il linguaggio Assembly

Giovanni Iacca
giovanni.iacca@unitn.it

Lezione basata su materiale preparato dal Prof. Luca Abeni

UNIVERSITÀ DEGLI STUDI DI TRENTO
Dipartimento di Ingegneria
e Scienza dell'Informazione

Linguaggio macchina ed Assembly

- CPU: “capisce” (e riesce ad eseguire) solo il suo linguaggio macchina
  - Sequenza di 0 e 1
  - Non proprio utilissimo per noi umani...
- Programmatore: scrive programmi in linguaggi di alto livello
  - C, C++, Java, ecc.
  - Non proprio comprensibili dalla CPU...
- Come riempire il gap?
- Compromessi fra sequenze di 0 e 1 ed alto livello?
  - Assembly!
  - Codici mnemonici invece di sequenze di 0 e 1 $\Rightarrow$ più facile da usare
  - Stretta corrispondenza fra istruzioni macchina ed Assembly $\Rightarrow$ sempre strettamente legato alla CPU

Programmare in Assembly

- Programma Assembly: file ASCII contenente descrizione testuale delle istruzioni
  - Deve essere compilato per essere eseguito dalla CPU
  - Assembler: compilatore da Assembly a linguaggio macchina
- Ad ogni istruzione Assembly corrisponde un'istruzione in linguaggio macchina
  - Salvo rare eccezioni (macro, label, riordinamento istruzioni...)
  - Le istruzioni Assembly dipendono dalla CPU
  - Programmi non portabili
- Linguaggio intermedio o linguaggio di programmazione?

Istruzioni Assembly

- Insieme delle istruzioni (sequenze di bit) riconosciute da una CPU: **I**ntruction **S**et **A**rchitecture (ISA)
  - Non una semplice lista di istruzioni
  - Sintassi e Semantica
  - Accesso ai dati
    - Registri (tipo e numero)
    - Modalità di accesso alla memoria (indirizzamento)
- L'ISA di una CPU ne definisce anche il linguaggio Assembly
  - In alcuni casi, più ISA per una singola CPU
  - Diversi linguaggi Assembly (Intel 32bit vs. Intel 64bit, ecc.)

Funzionamento di una CPU

- Struttura di un programma Assembly $\rightarrow$ deriva direttamente dal meccanismo di funzionamento di una CPU
  1. Fetch: preleva istruzione dalla memoria
     - Dove? Indirizzo memorizzato in apposito registro (Program Counter PC / Instruction Pointer IP)
  2. Decode: decodifica l'istruzione (per capire cosa fare)
  3. Execute: esegue l'istruzione (operazione aritmetico / logica, accesso alla memoria, ecc.)
- Programma $\equiv$ lista di istruzioni macchina eseguite (prevalentemente) in ordine sequenziale
  - Prevalentemente: esistono istruzioni di salto (per rompere l'esecuzione sequenziale)
  - Linguaggio di basso livello: salti, non cicli o selezioni!
- Tutto ciò si riflette sul linguaggio Assembly

Programmi Assembly

- Programma Assembly $\equiv$ lista di istruzioni eseguite (prevalentemente) in ordine sequenziale
  - Prevalentemente operazioni aritmetico / logiche
  - Talvolta operazioni di movimento dati
  - Ordine prevalentemente sequenziale: esistono operazioni di controllo del flusso (modificano il valore di PC / IP)
- Le istruzioni Assembly operano su dati (operandi dell'istruzione)
  - Operandi di operazione aritmetica o logica
  - Dati da muovere (ed indirizzi di memoria da / a cui muovere)
  - Nuovi valori per PC / IP
- Ancora: linguaggio di basso livello!

Registri

- Istruzioni Assembly: operano su dati
  - Immediati (costanti)
  - Contenuti in registri
  - Contenuti in memoria (varie modalità di indirizzamento)
- Registro a k bit: batteria di k flip-flop di tipo D
- Banco di registri utilizzabili da istruzioni Assembly
  - Cambia da CPU a CPU - definito da ISA
  - Registri general-purpose vs. registri specializzati
- In genere, numero di registri da 4 a 64
- Sintassi differente da Assembly ad Assembly
  - In alcuni casi nomi simbolici, in altri casi numeri

Istruzioni Assembly

- Istruzione Assembly: codice mnemonico seguito da eventuali operandi
  - Operandi: immediati, registri o in memoria
  - Alcune ISA pongono vincoli
- Tipi di istruzioni:
  - Aritmetico / logiche
  - Movimento dati
  - Controllo del flusso (salti, ecc.)
- Istruzioni aritmetico / logiche: specificano due operandi ed una destinazione (talvolta, destinazione implicita)
- Istruzioni che accedono alla memoria: varie modalità di indirizzamento
- Vincoli su istruzioni ed operandi: ISA CISC vs. RISC
  - RISC: semplifica l'implementazione della CPU
  - CISC: semplifica la scrittura di programmi Assembly

Esecuzione condizionale

- Alcune istruzioni vanno eseguite solo se determinate condizioni si verificano
  - Esempio: istruzioni di controllo del flusso
  - Necessario per implementare selezioni e cicli
  - In alcune ISA, altre istruzioni possono avere esecuzione condizionale (ARM: tutte!)
- Come esprimere queste condizioni?
  - Confronto fra valori di registri generici (general-purpose)
    - Come settare i valori di questi registri?
  - Basandosi sui valori di *flag* contenuti in uno speciale registro
    - Come settare i valori di questi flag?

Predicati in Assembly

- In alcune ISA, condizioni basate sul contenuto di registri generici
  - Esegui se due registri hanno contenuto uguale (o diverso)
  - Come implementare “esegui se il contenuto di un registro è maggiore del contenuto di un altro registro”?
  - Servono istruzioni di confronto!
- In altre ISA, esiste un registro *flag*
  - Vari flag settati da istruzioni aritmetiche e logiche
  - Flag settato se il risultato è 0, flag settato se il risultato è negativo, flag settato in caso di overflow, ecc.
- In alcune ISA, istruzione di confronto *set if less than* (o simile): confronta due registri general-purpose e setta un terzo registro in base al risultato
- In altre ISA, istruzione di confronto `cmp`: esegue sottrazione settando flag, ma scarta il risultato

Tipi di istruzioni Assembly

- Istruzioni Aritmetiche e Logiche
  - Implementate dalla ALU (Unitá Logica Aritmetica)
  - Operandi / Destinazione: solo nei registri o anche in memoria?
- Istruzioni di movimento dati
  - Sposta dati fra registri
  - Carica costanti (valori immediati) in registri / memoria
  - Carica dati da memoria a registri o viceversa
- Istruzioni di salto / controllo del flusso
  - Manipolano il contenuto di PC / IP
  - Necessaria esecuzione condizionale (selezioni e cicli)
  - In più: invocazione di subroutine e ritorno da subroutine
    - Necessario salvare il valore di PC / IP prima di cambiarlo...
    - Come?

Operandi e destinazioni

- Operazioni aritmetiche e logiche: generalmente due operandi ed una destinazione
  - Istruzioni Assembly con tre argomenti?
- Dove stanno gli operandi? Dove salvare il risultato (destinazione)?
- Due possibili “filosofie” diverse:
  1. Tutto nei registri (operandi immediati)
  2. Possibilità di avere operandi o destinazione in memoria
- Prima soluzione: implementazione CPU più semplice (e più elegante!)
- Seconda soluzione: linguaggio Assembly più potente e più semplice da usare
  - Talvolta, migliori performance
- La prima soluzione porta ad ISA RISC, la seconda ad ISA CISC

ISA CISC e RISC

- RISC: **R**educed **I**nstruction **S**et **C**omputer
  - ISA “più semplice” e regolare
  - Istruzioni aritmetico / logiche: no operandi o destinazione in memoria
  - Uniche istruzioni che accedono alla memoria: *load* e *store*
  - In generale, meno istruzioni e meno potenti
- CISC: **C**omplex **I**nstruction **S**et **C**omputer
  - Maggior numero di istruzioni; istruzioni più “potenti”
  - Tutte le istruzioni possono avere operandi (o destinazione) in memoria

Reduced Instruction Set Computer

- Obiettivo: semplificare al massimo la struttura della CPU
- Istruzioni aritmetico / logiche:
  `<opcode> <dst>, <arg1>, <arg2>`
  - `<dst>`, `<arg1>`: registri
  - `<arg2>`: registro o valore immediato
- Accessi alla memoria (sostanzialmente, solo due istruzioni):
  - `load <reg>, <memory location>`
  - `store <memory location>, <reg>`
- Conseguenza: servono più registri
- Codifica istruzioni: numero fisso di bit

Complex Instruction Set Computer

- Obiettivo: fornire istruzioni Assembly più potenti / flessibili / performanti
  - Istruzioni aritmetico / logiche: possono avere operandi e/o destinazione in memoria
  - Spesso ci sono comunque limiti / vincoli (Intel: al più un operando o la destinazione in memoria)
- Molte più istruzioni, con comportamenti più complessi
- Codifica istruzioni: numero variabile di bit
- Sintassi meno regolare
  - Esempio: per “salvare” alcuni bit di codifica, le istruzioni Intel hanno destinazione implicita (uguale al secondo argomento)

Meglio CISC o RISC?

- Domanda “filosofica”
  - Due tipi di ISA differenti, con obiettivi differenti
- Come in ogni cosa, gli estremismi non sono mai una buona idea...
- Soluzioni “di successo” spesso mescolano i due approcci
  - Intel: tipico esempio di ISA CISC, ma ha “rubato” alcune idee a RISC (numero di registri, `cmov`, ecc.)
  - ARM: ISA RISC “pragmatica” con modalità di indirizzamento più complesse, molte istruzioni (anche “complesse” e potenti), meno registri (16)
- Durante il corso vedremo tre ISA: MIPS (RISC “duro e puro”), Intel (CISC) ed ARM (RISC “pragmatico”)

Accessi alla memoria

- Istruzioni che accedono alla memoria:
  - Istruzioni `load` e `store` per RISC
  - Istruzioni generiche per CISC
  - In ogni caso, hanno argomento `<memory location>`
- Come si esprime `<memory location>`?
  Varie modalità di indirizzamento:
  1. Indirizzo di memoria costante espresso come valore immediato
  2. Indirizzo di memoria contenuto in un registro
  3. Indirizzo di memoria ottenuto shiftando (o comunque manipolando) il contenuto di un registro
  4. Combinazione dei metodi precedenti

Modalità di indirizzamento

- 1. Assoluto: indirizzo codificato nell'istruzione Assembly
  - Problematico per RISC... perché?
- 2. Indiretto: registro contenente indirizzo codificato nell'istruzione Assembly
  - Utile per implementare puntatori...
- 3. Base + Spiazzamento (2 + 1): indirizzo ottenuto sommando registro a valore immediato
- 4a. Base + Indice (2 + 3): indirizzo ottenuto sommando registro a registro scalato / shiftato
  - Utile per implementare array con elementi di dimensione > 1
- 4b. Base + Indice + Spiazzamento (1 + 2 + 3): indirizzo ottenuto sommando registro, registro scalato / shiftato e valore immediato

Indirizzamento vs. CISC / RISC

- Tradizionalmente, ISA RISC forniscono modalità di indirizzamento + semplici
  - Codificare indirizzi per indirizzamento assoluto è problematico (istruzioni codificate su numero fisso di bit)
- ISA CISC: indirizzamento con indice shiftato $\rightarrow$ permette di "risparmiare" istruzioni nell'accesso ad array
- Eccezione: ARM
  - Indirizzamento con indice shiftato
  - Pre-incremento / Post-incremento (utile nello scorrere array)

ISA e ABI

- ISA: definisce le istruzioni riconosciute dalla CPU ed i registri
  - Numero di registri
  - Registri general-purpose vs. registri specializzati
- Come usare i registri?
  - Quali registri usare per i dati e quali per gli indirizzi?
  - Invocazione di subroutine $\rightarrow$ i valori dei registri vengono cambiati?
  - Come si passano i valori dei parametri ed i valori di ritorno?
- Spesso questo non è forzato dall'hardware...
  $\Rightarrow$ ... convenzione software!
- **A**pplication **B**inary **I**nterface (ABI): insieme di convenzioni software che dicono come usare i registri
  - Esempio: convenzioni di chiamata

Convenzioni di chiamata

- Non fanno propriamente parte dell'architettura
  - Data una CPU / architettura, si possono usare 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!
- Convenzioni:
  - Come / dove passare i parametri: stack o registri?
  - Quali registri preservare?
  - Quando un programma invoca una subroutine, quali registri conterranno un certo valore al ritorno dalla subroutine?