Programmazione Assembly

Catena programmativa

Linguaggio Assembly

Istruzioni aritmetiche e logiche (esempi)

Pseudo istruzioni

Direttive

Claudia Raibulet

claudia.raibulet@unimib.it

La programmazione dei calcolatori

- **Linguaggio macchina (in binario)**
  - Linguaggio direttamente comprensibile dal calcolatore
  - Attivita' di programmazione piu' lunga
  - Facile commettere errori
- **Linguaggio assembler (Assembly)**
  - Rappresentazione simbolica del linguaggio macchina
  - Piu' comprensibile del linguaggio macchina (usa simboli, non sequenze di bit)
  - Tradotto dall'assemblatore in linguaggio macchina
  - Dalla forma simbolica dell'istruzione macchina al corrispondente formato binario
- **Linguaggi ad alto livello**
  - Tradotti dal compilatore in assembler

Linguaggi ad alto livello: vantaggi

- **Notazione vicina al linguaggio corrente e alta notazione algebrica (maggiore espressivita' e leggibilita')**
- **Incremento della produttivita'**
  - La programmazione e' svincolata dalla conoscenza dei dettagli architetturali della macchina utilizzata
- **Indipendenza dalle caratteristiche dell'architettura (processore) su cui il programma va eseguito (portabilita')**
  - Ideati non per essere compresi direttamente da macchine reali, ma da macchine astratte, in grado di effettuare operazioni di piu' alto livello rispetto alle operazioni dei processori reali
- **Permettono l'uso di librerie di funzionalita' gia' scritte (riusabilita' del codice)**

Assembler: vantaggi e svantaggi

- **Vantaggi: la dipendenza dall'architettura del calcolatore permette:**
  - Ottimizzare le prestazioni (maggiore efficienza)
  - Programmi (potenzialmente) piu' compatti
  - Massimo sfruttamento delle potenzialita' dell'hardware sottostante
  - Molto importante per programmare controller di processi e macchinari (e.g., real-time), o per apparati limitati (e.g., embedded computer, portatili)
- **Svantaggi: della programmazione**
  - Minore espressivita': e.g., strutture di controllo limitate
  - Necessario conoscere i dettagli dell'architettura
  - Mancanza di portabilita' su architetture diverse
  - Difficolta' di comprenssione
  - Lunghezza maggiore dei programmi

Compilatore

- **Un programma ad alto livello viene tradotto nel linguaggio assembler utilizzando un apposito programma: il compilatore**
- **Dopo la fase di compilazione, il programma scritto in linguaggio assembler viene tradotto in linguaggio macchina da un apposito programma: l'assemblatore**
- ***Osservazione:* spesso con il termine compilazione si indica l'intero processo di traduzione da linguaggio ad alto livello a linguaggio macchina (essendo l'assemblatore spesso integrato con il compilatore)**

Compiler, Assembler, Linker

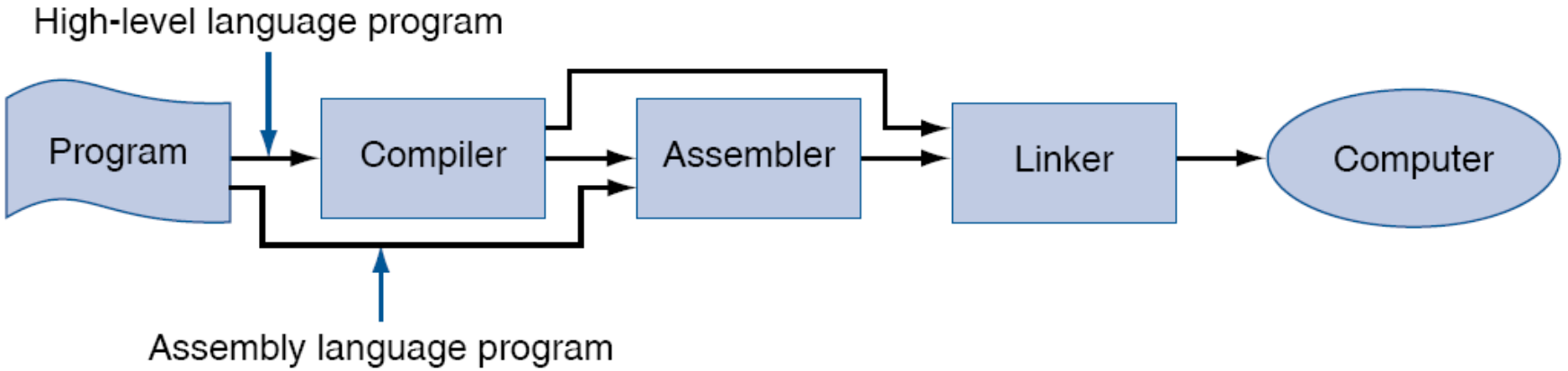

FIGURE A.1.6 Assembly language either is written by a programmer or is the output of a compiler.

Catena programmativa

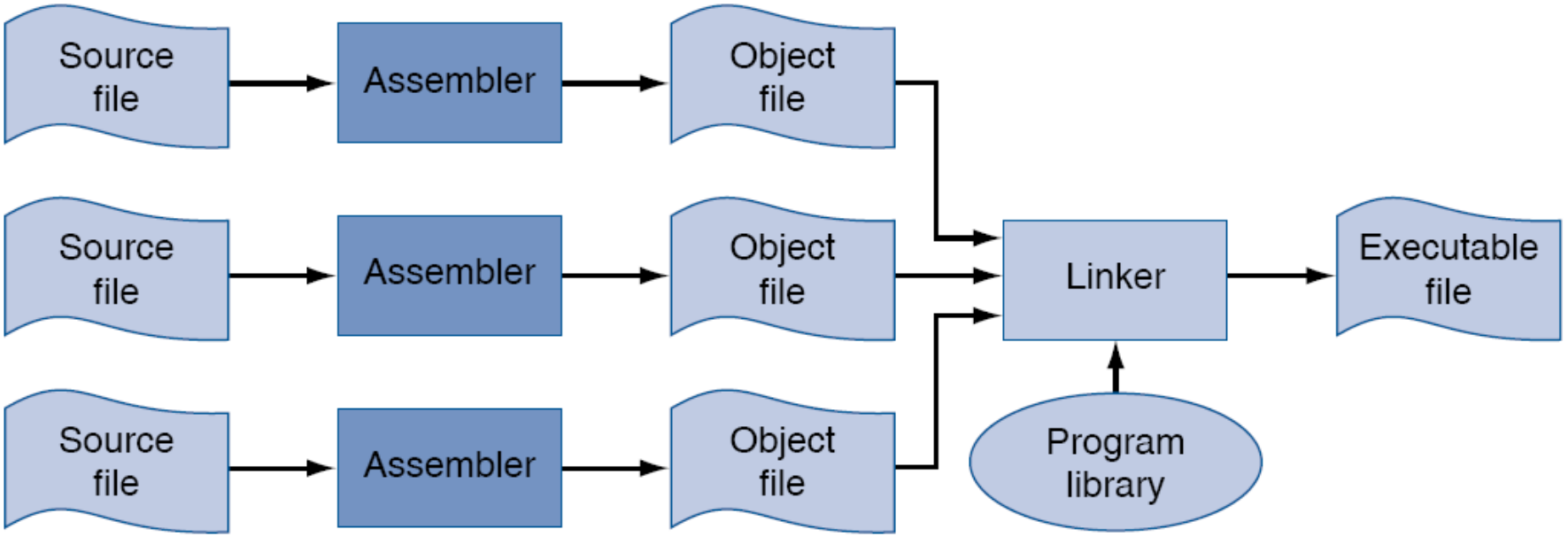

FIGURE A.1.1 The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

Assembler

- Converte un programma assembler (file sorgente) in linguaggio macchina (file oggetto)
- Gestisce le etichette
- Gestisce pseudoistruzioni
- Gestisce numeri in base diverse (e.g., binario, decimale, esadecimale)

Il processo di assemblaggio

- **L'assemblaggio e' un procedimento sequenziale che esamina, riga per riga, il codice sorgente Assembly, traducendo ciascuna riga in un'istruzione del linguaggio macchina**
- **Applicato modulo per modulo al programma e costituisce per ogni modulo la tabella dei simboli del modulo**
- **2 passi importanti:**
  - Traduce i codici mnemonici (simbolici) delle istruzioni nei corrispondenti codici binari
  - Traduce i riferimenti simbolici (variabili, registri, etichette di salto, parametri) nei corrispondenti indirizzi numerici
- **Poiche' le etichette di salto generano il problema dei riferimenti in avanti (ossia, riferimenti ad etichette successive o contenute in altri file), l'assemblatore deve leggere il programma sorgente due volte**
- **Ogni lettura del programma sorgente è chiamata passo e l'assemblatore è chiamato traduttore a due passi.**
- **Ogni modulo assemblato di default parte dall'indirizzo 0; In sistemi dotati di meccanismi di memoria virtuale tali indirizzi sono indirizzi virtuali**

Assembler – Tabella dei Simboli

- **Contiene i riferimenti simbolici presenti nel modulo da tradurre e, al termine del primo passo, conterrà gli indirizzi numerici di tutti i simboli, tranne quelli esterni al modulo in esame**
- **Per le etichette associate a direttive dell'assemblatore che definiscono costanti simboliche nella tabella dei simboli viene creata la coppia < etichetta, valore > e in ogni istruzione che fa riferimento al simbolo viene sostituito il valore**
- **Per etichette che definiscono variabili (spazio di memoria + eventuale inizializzazione), l'assemblatore riserva spazio, eventualmente inizializza la zona di memoria e crea nella tabella la coppia < etichetta, indirizzo >. In ogni istruzione che fa riferimento al simbolo viene sostituito l'indirizzo**
- **Nelle etichette presenti nelle istruzioni di salto, l'assemblatore deve generare un riferimento all'indirizzo dell'istruzione destinazione di salto**

Assembler – Tabella dei Simboli

- **Osservazioni: le etichette esterne (global) al modulo possono essere usate da moduli esterni; le etichette interne (local) sono visibili solo all'interno di un modulo**
- **Le etichette esterne a un modulo rimangono non risolte -> l'assember non e' disturbato da questo aspetto**

Linker (Link editor)

- **Inserisce in memoria in modo simbolico il codice e i moduli dati**
- **Determina gli indirizzi dei dati e delle etichette che compaiono nelle istruzioni**
- **Corregge i riferimenti interni ed esterni e risolve i riferimenti in sospeso (a etichete esterne)**
- **Genera il file eseguibile**

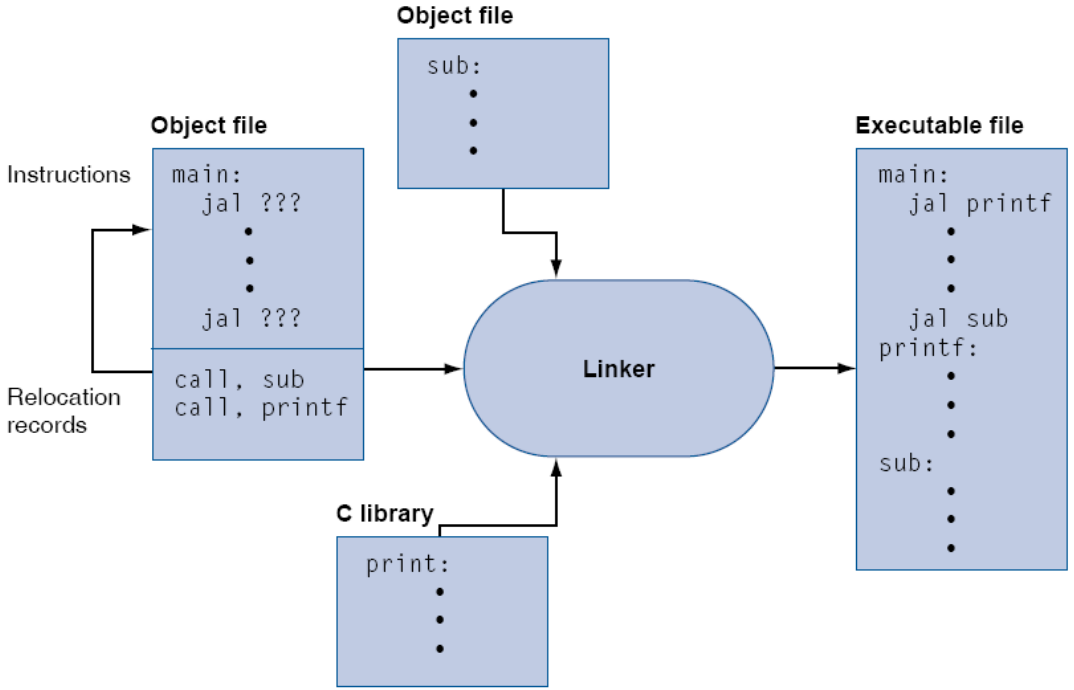

FIGURE A.3.1 The linker searches a collection of object files and program libraries to find nonlocal routines used in a program, combines them into a single executable file, and resolves references between routines in different files.

Loader

- Lettura dell'intestazione del file eseguibile per determinare la lunghezza del segmento di testo e del segmento dati
- Creazione di uno spazio di indirizzamento sufficiente a contenere testo e dati
- Copia delle istruzioni e dati dal file eseguibile in memoria
- Copia nello stack degli eventuali parametri passati al programma principale
- Inizializzazione dei registri e impostazione dello stack pointer affinche' punti alla prima locazioni libera
- Salto a una procedura di startup la quale copia i parametri nei registri argomento e chiama la procedura principale del programma
- Quando la procedura principale restituisce il controllo, la procedura di startup termina il programma con una chiamata alla funzione di sistema exit

Convenzioni assembly: nomi e usi dei registri

Nome Simbolico	Numero	Uso
$zero	0	Costante 0
$at	1	Assembler temporary
$v0-$v1	2-3	Functions and expressions evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Saved Temporaries
$t8-$t9	24-25	Temporaries
$k0-$k1	26-27	Reserved for OS kernel
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

- Usati da assembler, compilatore, sistema operativo
- Secondo specifiche convenzioni (ne parliamo in seguito)
- …da trattare con cautela se si programma in assembly!!!

Direttive all'assemblatore (Meccanismi misteriosi)

- **NON corrispondono a istruzioni macchina**
- **Sono indicazioni date all'assembler per consentirgli di**
  - associare etichette simboliche a indirizzi
  - allocare spazio di memoria per le variabili
  - decidere in quali zone di memoria allocare istruzioni e dati
  - ...
- **Esempi di direttive**
- **.data <addr>**
  - quel che segue va nel segmento dati (eventualmente dall'indirizzo addr)
- **.byte b1,....,bn**
  - inizializza i valori in byte successivi
- **.word w1,....,wn**
  - inizializza i valori in word successive
- **.text <addr>**
  - quel che segue va nel segmento text (programma) (eventualmente dall'indirizzo addr)

Esempio

```
        .data
item:   .word 1
        .text
        .globl main     #meccanismo misterioso
main:   la      $t0, item       # carico l'indirizzo di memoria
                                # della variabile item in $t0
        lw      $t1, 0($t0)     # carico in $t1 il valore che si
                                # trova all'indirizzo di memoria
                                # 0 piu' il valore che sta nel registro $t0
```

Memoria

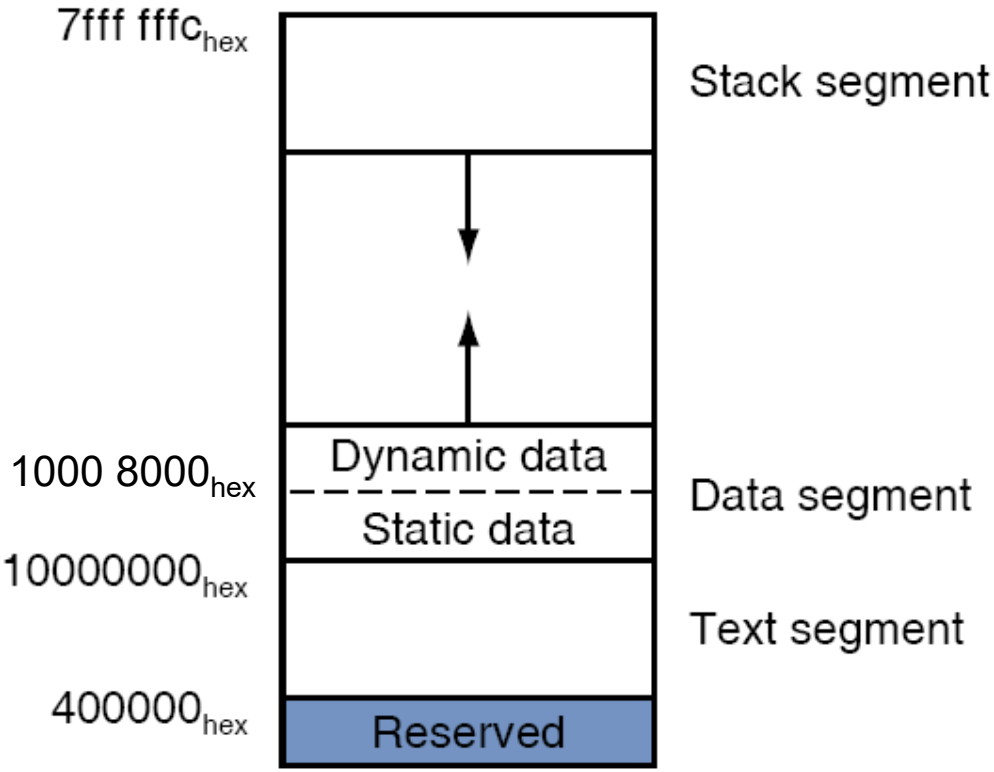

FIGURE A.5.1 Layout of memory.

Istruzioni aritmetiche e logiche (esempi)

- **add rd, rs, rt**
  - addizione
  - rs + rt -> rd
  - (con overflow...ne parliamo in seguito...)
- **addi rd, rs, imm**
  - addizione immediata
  - sign-extended imm + rs -> rd
  - (con overflow)
- **and rd, rs, rt**
  - and bit a bit di rs e rt -> rd
- **or rd, rs, rt**
  - or bit a bit (logical or) di rs e rt -> rd
- **ori rt, rs, imm**
  - or bit a bit (logical or) di rs e zero-extended imm -> rt
- **sll rd, rt, shamt**
  - shift left rt della distanza shamt -> rd

Manipolazione di costanti

- **lui rt, imm**
  - load upper immediate
  - immediate (di 16 bit!!!) -> upper half word di rt
  - I 16 bit bassi di rt sono 0
- **...e se si vuole caricare una costante di 32 bit?**
- **esempio: caricare in $s0 il valore 4000000**

  0000 0000 0011 1101 0000 1001 0000 0000

lui $s0, 61 (oppure: lui $s0, 0x003d)

contenuto di $s0: 0000 0000 0011 1101 0000 0000 0000 0000

ori $s0, $s0, 2304 (oppure: ori $s0, 0x0900)

contenuto di $s0: 0000 0000 0011 1101 0000 1001 0000 0000

Pseudoistruzioni

- ***Pseudo istruzione:***
  - istruzione assembly che non ha una corrispondente istruzione macchina
  - tradotta dall'assembler in una sequenza di istruzioni:
- **Esempio:**
- **li rdest, imm**
  - load immediate
  - caricare una costante di 32 bit nel registro rdest

  li $s0, 4000000
- **Tradotta dall'assemblatore nella sequenza della slide precedente**
- **Esempio 2:**

  move $t0, $t1
- **Tradotta dall'assemblatore in**

  add $t0, $zero, $t1

Osservazioni

- **I programmi sono immagazzinati in memoria insieme ai dati -> concetto di programma memorizzato**
- **In memoria abbiamo istruzioni e operandi che devono essere entrambi trasferiti dalla memoria al processore**
- **Le istruzioni non sono di per se' distinguibili rispetto agli altri tipi di informazione in memoria: il processore interpreta se una configurazione di bit rappresenta un dato o un'istruzione**

MIPS

- **Progettata e sviluppata a Stanford (USA)**
- **MIPS Technologies**
- **https://www.mips.com/**

- **Utilizzata da:**
  - Sony (Playstation, Playstation 2)
  - Nintendo 64
  - Router CISCO
  - Stampanti
  - Macchine fotografiche digitali
  - DVD,
  - TV al plasma
  - ...

- **Rappresenta un buon modello architetturale per la didattica in quanto semplice da comprendere**

Riferimenti principali

- **Patterson – Hennessy, Computer Organization and Design, Morgan Kaufmann**
  - Capitolo 2, Sezioni da 1 a 7, Sezione10, parte iniziale della 12
  - Appendice B, almeno B.1,B.2, B.9, **B.10**
- **...ma non è vietato dare una scorsa almeno all'indice e alle introduzioni di tutti i capitoli**
- **Appendice B (A nella vecchia edizione)**

Esercizio

- **Si chiede di scrivere il programma assembly che legge 3 valori word a, b, e c memorizzati nella memoria all'etichetta "valori" nei registri $t0, $t1, $t2 e calcola:**
  - La somma dei tre valori in $s0 se il primo valore e' positivo
  - Il prodotto dei tre numeri in $s0 se il primo valore e' negativo
  - L'AND del secondo e del terzo numero se il primo e' uguale a zero.
- **Si chiede di memorizzare il risultato in memoria nella locazione subito dopo il terzo valore.**

Soluzione

```
            .data
valori:     .word 0, 5, 2
            .word 0
            .text
            .globl main

main:       la $t7, valori

            lw $t0, 0($t7)
            lw $t1, 4($t7)
            lw $t2, 8($t7)

            bgt $t0, $zero, than # se $t0>0
            blt $t0, $zero, else  # se $t0<0
            and $s0, $t1, $t2
            j fine

else:       #calcola il prodotto
            mul $s0, $t0, $t1
            mul $s0, $s0, $t2
            j fine

than:       #calcola la somma
            add $s0, $t0, $t1
            add $s0, $s0, $t2
            j fine

fine:       sw $s0, 12($t7)
```