

Livello Software

Architettura degli elaboratori AA 2023/24

Corso di Laurea Triennale in Informatica

Massimo Orazio Spata

massimo.spata@unict.it

Dipartimento di Matematica e Informatica



UNIVERSITÀ
degli STUDI
di CATANIA

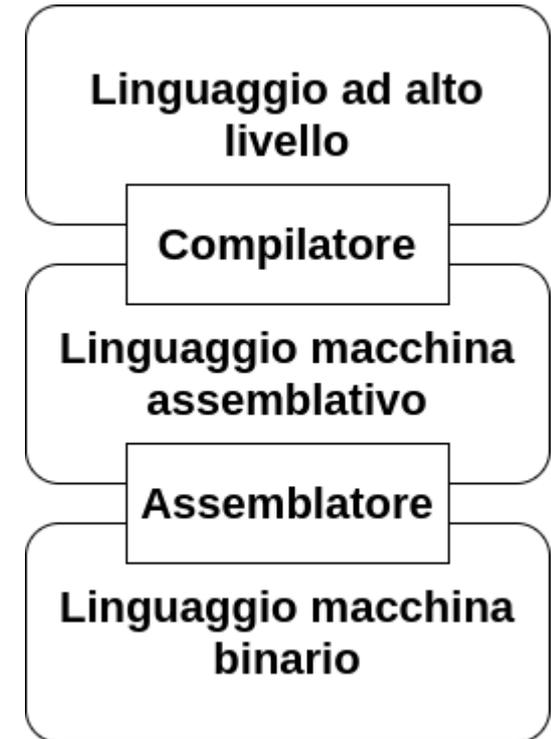
Come si programma?

• Il programmatore scrive i programmi in **LINGUAGGIO ASSEMBLATIVO (ASSEMBLY)**

• Il programma assembler viene tradotto in sequenze binarie dall'**ASSEMBLATORE (ASSEMBLER)**

• Linguaggi ad alto livello (C, C++, etc.) ancora più espressivi

• Il **COMPILATORE** traduce il codice ad alto livello in codice assembler

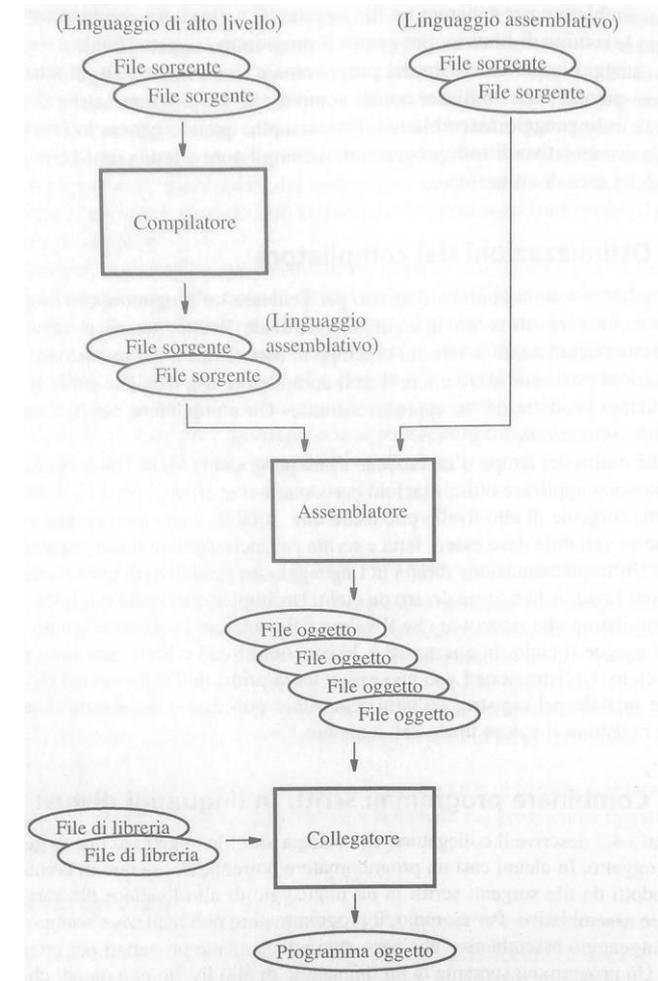


Generazione programma oggetto

• Il **COMPILATORE** trasforma una serie di file sorgenti di linguaggio ad alto livello in file sorgenti assembly

• L'**ASSEMBLATORE** trasforma una serie di file sorgenti assembly (generati dal compilatore o scritti direttamente da un programmatore) in file oggetto

• Il **LINKER** collega assieme vari file oggetto e file di libreria in un unico programma oggetto (un file oggetto potrebbe usare dei sottoprogrammi definiti in altri file oggetto o di libreria)



- L'assemblatore traduce un file sorgente, scritto in assembly, in un file oggetto, scritto in codice macchina binario
- Per generare il file oggetto l'assemblatore deve eseguire i seguenti passi:
 - Generare la codifica binaria delle istruzioni espresse in assembly (codice operativo e operandi)
 - Riconosce le direttive di assemblatore per l'allocazione di memoria mettendo queste informazioni nell'header del file oggetto
 - Riconosce le direttive che assegnano nomi a costanti (i.e. EQU) e sostituisce il valore binario ad ogni occorrenza dei nomi nel codice
 - Sostituisce il valore binario ad ogni occorrenza di etichette di indirizzi relativi nel codice

- Man mano che incontra etichette e dichiarazioni di costanti, l'assemblatore tiene traccia dei nomi e dei valori corrispondenti nella **tabella dei simboli**
- L'assemblatore sostituisce ogni occorrenza di un nome con il valore indicato nella **tabella dei simboli**
- **Ma cosa accade se il simbolo viene dichiarato dopo essere usato?**
(etichette dei salti in avanti per esempio)
- Si usa un metodo di assemblaggio a 2 passate:
 - Passo 1: si scorre il codice sorgente raccogliendo in tabella tutti i simboli e i rispettivi valori numerici
 - Passo 2: si scorre nuovamente il codice sorgente sostituendo i simboli con i valori in tabella e generando il codice oggetto finale

- File sorgente, file oggetto e dati sono inizialmente memorizzati nella memoria secondaria (su disco)
- Per essere eseguito dal processore, un programma oggetto e i suoi dati devono essere trasferiti in memoria centrale
- Il **LOADER** è il programma dedito a caricare il programma oggetto in memoria, attivato da una richiesta di esecuzione del programma oggetto da parte dell'utente
- Il **Loader** deve eseguire le seguenti operazioni:
 - Leggere informazioni quali lunghezza del programma e locazione di caricamento dall'header del file oggetto
 - Caricare il programma in memoria sulla base di tali informazioni
 - Saltare alla prima istruzione del programma da eseguire

- Nella maggior parte dei casi un programma è distribuito in più file sorgente
- In un file sorgente si possono avere chiamate a sottoprogrammi dichiarati in altri file sorgente
- In questi casi l'assemblatore genera un file oggetto incompleto (con riferimenti a **nomi esterni**) per ogni file sorgente
- Il **LINKER** genera un file oggetto completo combinando più file oggetto separati e risolvendo i riferimenti a nomi esterni
- Ogni file oggetto deve contenere una **lista di nomi esterni usati** e una **lista delle etichette da esportare**

- Le **Librerie** sono file che raggruppano file oggetto con sottoprogrammi utilizzabili da altri programmi
- I file di libreria sono creati dal programma di utilità chiamato **ARCHIVER**
- Nel file di libreria sono inserite le informazioni per permettere al Linker di risolvere i riferimenti a nomi esterni in altri programmi
- I file di libreria usati devono essere specificati all'invocazione del Linker
- I file oggetto rilevanti della libreria saranno inclusi nel programma oggetto finale da parte del Linker

- Normalmente i programmi vengono scritti in linguaggi ad alto livello molto espressivi
- Il **COMPILATORE** trasforma un file sorgente scritto in linguaggio ad alto livello in un file scritto in assembly
- Il compilatore automatizza molti compiti del programmatore assembly (gestione delle aree di attivazione per esempio)
- Un compilatore che riorganizza le istruzioni per ottimizzare il codice viene detto **OTTIMIZZANTE**
- Un programma ad alto livello può chiamare sottoprogrammi presenti in altri file assembly o scritti in altri linguaggi (il linker gestirà i collegamenti)

- Prendiamo come esempio un programma che legge caratteri da tastiera e li visualizza a video tramite scansione
- In questo programma si accede solo a registri di I/O (stesso spazio di indirizzamento delle locazioni di memoria)
- Si può scrivere tale programma interamente in linguaggio C (bisogna conoscere gli indirizzi dei registri di I/O)
- Il programma C è notevolmente più corto dell'equivalente in assembly (la gestione dei registri del processore è automatizzata)

Assembly - I/O tramite scansione

KBD_DATA	EQU	0x4000	Registro dati della tastiera (8 bit)
KBD_STATUS	EQU	0x4004	Registro di stato della tastiera (il bit 1 è la condizione di stato KIN)
DISP_DATA	EQU	0x4010	Registro dati dello schermo (8 bit).
DISP_STATUS	EQU	0x4014	Registro di stato dello schermo (il bit 2 è la condizione di stato DOUT)
	Move	R2, #KBD_DATA	Puntatore all'interfaccia del dispositivo tastiera
	Move	R3, #DISP_DATA	Puntatore all'interfaccia del dispositivo schermo
CICLO_KBD :	LoadByte	R4, 4(R2)	Controlla se c'è in ingresso un carattere dalla tastiera
	And	R4, R4, #2	
	Branch_if_[R4]=0	CICLO_KBD	
	LoadByte	R5, (R2)	Leggi il carattere ricevuto
CICLO_DISP :	LoadByte	R4, 4(R3)	Controlla se lo schermo è pronto per un nuovo carattere
	And	R4, R4, #4	
	Branch_if_[R4]=0	CICLO_DISP	
	StoreByte	R5, (R3)	Scrivi il carattere ricevuto sullo schermo
	Branch	CICLO_KBD	

C - I/O tramite scansione

```
/* Definizione di indirizzi di registri */
#define KBD_DATA      (volatile char *) 0x4000
#define KBD_STATUS    (volatile char *) 0x4004
#define DISP_DATA     (volatile char *) 0x4010
#define DISP_STATUS   (volatile char *) 0x4014

void main()
{
    char ch;

    /* Trasferisci i caratteri */
    while (1) {
        while ((*KBD_STATUS & 0x2) == 0); /* Ciclo senza fine */
        ch = *KBD_DATA; /* Attendi un nuovo carattere */
        while ((*DISP_STATUS & 0x4) == 0); /* Leggi il carattere dalla tastiera */
        *DISP_DATA = ch; /* Attendi che lo schermo sia pronto */
    } /* Trasferisci il carattere allo schermo */
}
```

- In alcuni casi è necessario inserire delle istruzioni assembly all'interno del codice C (per esempio per l'accesso a registri di controllo del processore)
- Alcuni compilatori C permettono di inserire funzioni assembly nel codice attraverso delle direttive al compilatore: **asm("codice-assembly")**
- È spesso possibile definire le routine di servizio di interruzione in C usando la parola chiave **interrupt** all'inizio di una definizione di funzione: **interrupt void intserv() {...}**

Vediamo ora come esempio un programma che riceve dei caratteri da tastiera tramite interruzione e li stampa su schermo

Assembly - I/O tramite interruzione

IVETT	EQU	0x20	Vettore per routine di servizio d'interruzione
KBD_DATA	EQU	0x4000	Registro dati della tastiera (8 bit)
KBD_STATUS	EQU	0x4004	Registro di stato della tastiera (il bit 1 è la condizione di stato KIN)
KBD_CONT	EQU	0x4008	Registro di controllo della tastiera (il bit 1 è la condizione di stato KIE)
DISP_DATA	EQU	0x4010	Registro dati dello schermo (8 bit)
DISP_STATUS	EQU	0x4014	Registro di stato dello schermo (il bit 2 è la condizione di stato DOUT)

Programma principale

MAIN:	Move	R2, #KBD_DATA	Puntatore all'interfaccia della tastiera
	Move	R3, #0x2	
	StoreByte	R3, 8(R2)	Configura la tastiera per segnalare interruzioni
	Move	R2, #IVETT	Puntatore al vettore delle interruzioni
	Move	R3, #INTSERV	Inizio della routine di servizio d'interruzione
	Store	R3, (R2)	Inizializza il vettore delle interruzioni
	Move	R2, #0x2	Abilita il processore a ricevere interruzioni da tastiera
	MoveControl	IENABLE, R2	
	Move	R2, #0x1	Poni a 1 il bit che abilita le interruzioni per il processore

CICLO :

MoveControl	PS, R2	
Branch	CICLO	Ciclo di attesa continua

Routine di servizio d'interruzione

INTSERV:	Subtract	SP, SP, #8	Salva i registri
	Store	R2, 4(SP)	
	Store	R3, (SP)	
	Move	R2, #KBD_DATA	Puntatore all'interfaccia della tastiera
	LoadByte	R3, (R2)	Leggi il prossimo carattere
	Move	R2, #DISP_DATA	Puntatore all'interfaccia dello schermo
	StoreByte	R3, (R2)	Scrivi il carattere ricevuto sullo schermo
	Load	R2, 4(SP)	Ripristina i registri
	Load	R3, (SP)	
	Add	SP, SP, #8	
	Return-from-interrupt		

C - I/O tramite interruzione

```
#define IVECT      (volatile unsigned int *) 0x20
#define KBD_DATA  (volatile char *) 0x4000
#define KBD_CONT  (volatile char *) 0x4008
#define DISP_DATA (volatile char *) 0x4010
#define DISP_STATUS (volatile char *) 0x4014

interrupt void intserv(); /* Dichiarazione anticipata di funzione */

void main()
{
    /* Inizializzazioni per trasferimenti di caratteri tramite interruzioni */
    *KBD_CONT = 0x2; /* Abilita le interruzioni da tastiera */
    *IVETT = (unsigned int) &intserv; /* Inizializza il vettore delle interruzioni */
    asm ("Subtract SP, SP, #4"); /* Salva il registro R2 */
    asm ("Store R2, (SP)");
    asm ("Move R2, #0x2"); /* Abilita il processore a ricevere interruzioni da tastiera */
    asm ("MoveControl IENABLE, R2");
    asm ("Move R2, #0x1"); /* Abilita le interruzioni per il processore */
    asm ("MoveControl PS, R2");
    asm ("Load R2, (SP)"); /* Ripristina il registro R2 */
    asm ("Add SP, SP, #4");

    while (1) /* Ciclo continuo */
    {
        /* Trasferisci i caratteri usando la routine di servizio d'interruzione */
    }

    interrupt void intserv() /* La parola chiave indica al compilatore di trattare la funzione come routine di servizio d'interruzione */
    {
        *DISP_DATA = *KBD_DATA; /* Trasferisci un carattere */
        /* Il compilatore inserirà l'istruzione di rientro da interruzione alla fine della funzione */
    }
}
```

- Il compilatore è in grado di rilevare errori sintattici e nomi sconosciuti nel codice sorgente, ma non errori di programmazione (**bug**)
- Il **DEBUGGER** è un programma che ci permette di eseguire il programma oggetto ed interrompere la sua esecuzione in qualsiasi istante per valutarne il corretto funzionamento, controllando lo stato della memoria e dei registri
- Il Debugger può essere eseguito in due diverse modalità:
 - **trace mode:** il programma viene eseguito passo-passo, interrompendosi dopo ogni istruzione
 - **breakpoint:** l'esecuzione del programma si interrompe in punti di osservazione specifici definiti dal programmatore

Passi di esecuzione in Trace mode:

- Si genera un'eccezione al termine dell'esecuzione di ogni istruzione del programma
- Il Debugger viene lanciato come routine di servizio dell'istruzione
- Il programmatore controlla lo stato del programma
- Una volta che il programmatore seleziona il comando per continuare l'esecuzione viene effettuato un rientro dall'interruzione e viene eseguita l'istruzione successiva

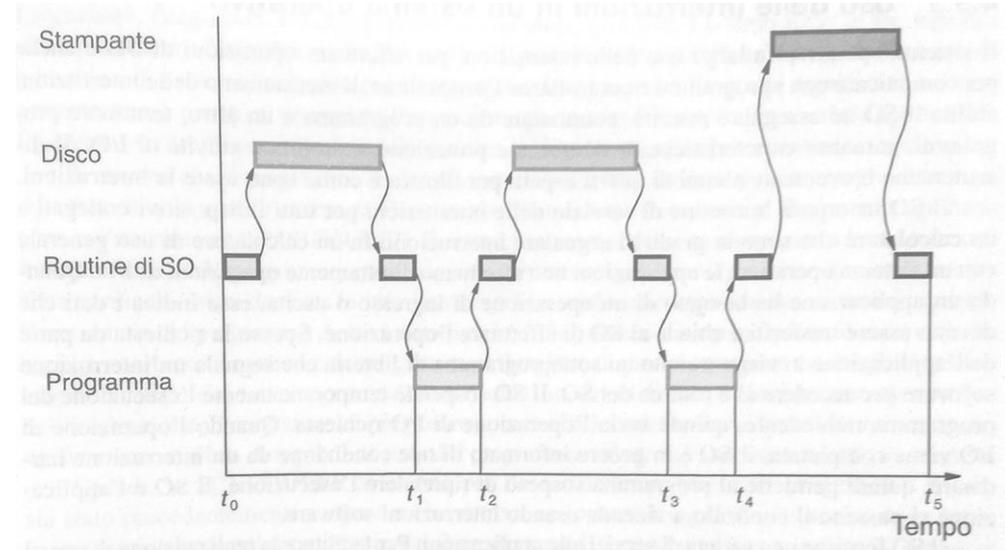
Passi di esecuzione in usando i breakpoint:

- Quando il Debugger è in esecuzione, il programmatore può scegliere dei punti di osservazione (**breakpoint**) dove interrompere il programma
- Il Debugger sostituisce e mette da parte le istruzioni in corrispondenza dei breakpoint con speciali interruzioni software (**Trap**)
- Il programma viene eseguito normalmente fino ad arrivare alla prima Trap, dove l'esecuzione passa al Debugger
- Una volta che il programmatore seleziona il comando per continuare l'esecuzione il Debugger riprende l'esecuzione del programma e reinserisce la trap

- Il **Sistema Operativo (SO)** gestisce il coordinamento generale di tutte le attività del calcolatore (esecuzione concorrente di programmi, gestione degli I/O, gestione dell'accesso alla memoria, gestione dell'interfaccia utente, etc.)
- Il **SO** è formato da un insieme di routine essenziali che risiedono nella memoria centrale e un insieme di programmi di utilità che risiedono su disco e vengono caricati in memoria centrale per essere eseguiti
- Durante l'inizializzazione del sistema, un processo di avvio (**boot-strapping**) viene usato per caricare in memoria una porzione iniziale del SO

Esempio di gestione programmi

- Caso semplice in cui ci sia un solo programma in esecuzione che richiede la lettura di dati dal disco e la stampa degli stessi
- Il processore salterà ripetutamente dall'esecuzione di routine del sistema operativo, di routine di I/O e del programma
- Nel caso di più programmi da eseguire contemporaneamente è possibile parallelizzare (virtualmente) il processo
- Sistemi operativi capaci di eseguire più programmi contemporaneamente sono chiamati **concorrenti** o **multitasking**



- Per comunicare con programmi, memoria e periferiche di I/O, il SO fa uso di un sistema di interruzioni
- Il SO fornisce una serie di routine di servizio di interruzioni software per svariati compiti, ognuna con il proprio vettore di interruzione
- Per gestire programmi concorrenti il SO ne divide l'esecuzione in quanti di tempo (**time slicing**)
- Un contatore lancia un interruzione ogni quanto di tempo τ , e lancia la routine **SCHEDULER** per scegliere il prossimo programma da eseguire
- I programmi possono trovarsi in 3 stati: **RUNNING**, **RUNNABLE** e **BLOCKED**