Architettura degli Elaboratori, a.a. 2006-07

Parte Seconda: guida allo studio e integrazioni

La Seconda Parte del corso, che riguarda principalmente il livello assembler e principi di architettura firmware di calcolatori general-purpose:

- inizia con una **introduzione al modello di calcolatore general-purpose "a programma memorizzato"** ("modello di Von Neumann"):
  - *sez. 1, 2* di queste note;
- studia le **caratteristiche del livello assembler** e si concentra su **uno specifico linguaggio assembler di tipo RISC (Reduced Instruction Set Computer) e sua utilizzazione nella compilazione di programmi:**
  - Cap. V delle Dispense, sez. 1, 2;
  - *sez. 3* di queste note per quanto riguarda una introduzione al concetto di **memoria virtuale e spazio di indirizzamento**;
  - Cap. V delle Dispense, sez. 3;
  - *sez. 4* di queste note per un esempio completo;
  - *sez. 5* di queste note per testi di esercizi.
- studia una prima **architettura di un calcolatore general-purpose**, capace di supportare (interpretare) il set di istruzioni assembler del Cap. V, e studia come effettuare la **valutazione delle prestazioni di calcolatori e di programmi**:
  - Cap. VII delle Dispense, eccetto sez. 2.3.2 e 2.3.3;
  - *sez. 6* di queste note per testi di esercizi;
- è accompagnata dalla **seconda Esercitazione di Verifica Intermedia**.

1. Modello di calcolatore general-purpose a programma memorizzato

Nell'architettura firmware di un calcolatore general-purpose, un ruolo chiave è svolto dal *Processore*: il microprogramma di questa unità di elaborazione (che, come per qualunque unità, definisce il suo funzionamento è la sua struttura) è *l'interprete del linguaggio assembler*, permettendo quindi *l'esecuzione del programma in versione eseguibile*.

Ci si potrebbe chiedere "perché il livello firmware è un interprete e non un compilatore": in effetti, si può dimostrare che, nella strutturazione a livelli di un sistema di elaborazione, almeno il livello più basso deve essere un interprete. Nel caso della strutturazione tipica (vedi Cap. I della Dispensa), esistono almeno due interpreti: quello a livello firmware (che interpreta l'assembler) e quello a livello hardware (che interpreta il firmware). Al più, si potrebbe pensare di compilare l'assembler in firmware, con che il vero "linguaggio macchina" diverrebbe il microlinguaggio, rendendo di fatto inutile il livello assembler; questa soluzione, che è stata adottata con successo nel passato, non è più conveniente alla luce dell'evoluzione della tecnologia VLSI (occorrerebbe realizzare processori "microprogrammabili"). D'altra parte, specie se le istruzioni assembler sono semplici (*approccio Risc*), i microprogrammi risultano notevolmente ottimizzati anche in un approccio interpretato.

Con la terminologia del Cap. III della Dispensa, le istruzioni assembler sono le "operazioni esterne" dell'unità di elaborazione Processore.

Una volta stabilito di realizzare di realizzare il Processore come interprete del linguaggio assembler, discende che, nell'architettura firmware del calcolatore, occorre prevedere anche una unità *Memoria Dati*, nella quale fare risiedere le strutture dati riferite dal programma eseguibile, ed un insieme di *Unità di Ingresso/Uscita* per trasferire dati a/da tale memoria (dischi, monitor, tastiere, mouse, stampanti, interfacce di rete, ecc).

Inoltre, occorre prendere una decisione cruciale relativamente all'architettura complessiva del calcolatore:

- quale modello di programmazione segue il linguaggio assembler?
- come vengono fatte pervenire al Processore le richieste di esecuzione delle istruzioni assembler (operazioni esterne), in modo da rispettarne efficientemente l'ordinamento a tempo di esecuzione?

La risposta al primo quesito comporta una risposta al secondo. Ad esempio, avremmo architetture completamente diverse a seconda che il modello di programmazione del linguaggio assembler fosse puramente funzionale oppure imperativo. La scelta universalmente adottata ormai da molti anni è la seguente:

- il modello di programmazione a livello assembler è *imperativo* (per comprenderne a fondo i motivi, occorre studiare aspetti avanzati di Architettura degli Elaboratori, come nel corso di Architetture Parallele e Distribuite).

Questa scelta conduce al così detto *modello di architettura a programma memorizzato*, o *modello di Von Neumann*:

- il programma assembler da eseguire (da interpretare da parte del Processore) risiede in una memoria accessibile al Processore, detta *Memoria Principale* del calcolatore.

Questo comporta che le *istruzioni del programma assembler, rappresentate in binario* (come stringhe di bit), siano *memorizzate come parole di memoria*, che possono essere lette dal Processore nel giusto ordine; cioè, *è il Processore stesso (il suo microprogramma) che reperisce una istruzione alla volta dalla memoria e provvede alla sua esecuzione (interpretazione).* Più in dettaglio:

i) una volta letta dalla memoria, ogni istruzione verrà riconosciuta univocamente, attraverso un suo campo *Codice Operativo*, in modo che il microprogramma del Processore possa svolgere quelle azioni che sono necessarie per l'esecuzione dell'istruzione stessa;

ii) completata una istruzione, il Processore deve poter individuare l'istruzione successiva. A questo scopo il Processore dispone di un registro, detto *Contatore Istruzioni* (IC), che contiene l'indirizzo dell'istruzione da eseguire; il contenuto di IC verrà modificato, durante il (alla fine del) microprogramma di ogni istruzione, in modo da portarlo ad assumere il valore dell'indirizzo della prossima istruzione da leggere dalla memoria;

iii) tra le azioni da effettuare durante l'interpretazione di una istruzione, ci possono essere anche letture o scritture di dati del programma residenti nella *Memoria Dati*. In una architettura a programma memorizzato, questa viene fatta *coincidere logicamente con la Memoria Principale*. Ne consegue che, nel modello di Von Neumann, *concettualmente non c'è distinzione tra istruzioni e dati*: in effetti, entrambe le informazioni sono dati utilizzati dal microprogramma-interprete e reperiti in una memoria accessibile al Processore; un particolare tipo di dato è "l'istruzione", così come in una unità specializzata (vedi Cap. III della Dispensa) i messaggi in ingresso contengono informazioni sia sull'operazione esterna che sui dati ad essa associati.

L'architettura firmware di un elaboratore general-purpose assume allora la forma schematica della fig. 1

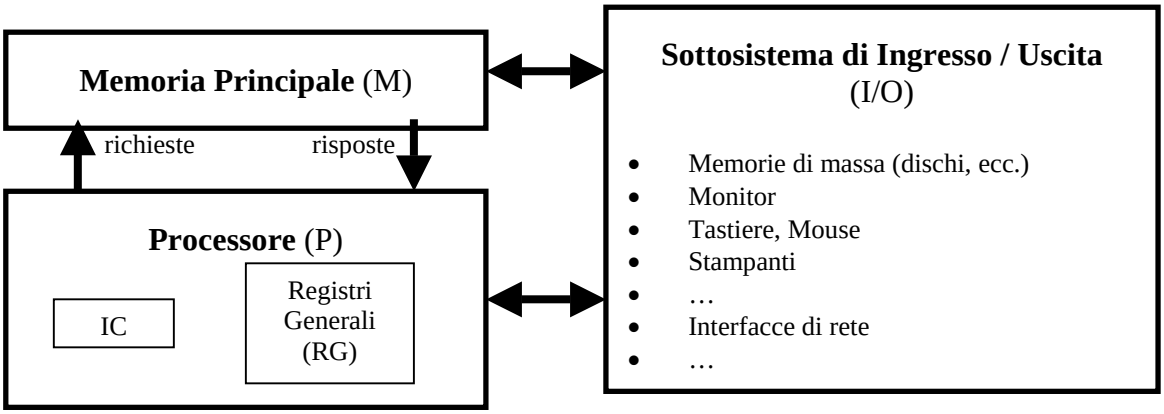


Fig. 1

Note sulla fig. 1:

(1) *richieste*: richieste di accesso alla memoria da parte di P; possono essere richieste di lettura di una istruzione all'indirizzo IC, oppure richieste di lettura o di scrittura di una dato all'indirizzo generato durante l'esecuzione del microprogramma;

(2) *risposte*: parole lette dalla memoria (istruzioni o dati) in risposta ad una richiesta di P, più informazioni sull'esito dell'accesso;

(3) nel Processore, oltre al registro contatore istruzioni IC (fondamentale in qualunque realizzazione del modello di Von Neumann), sono indicati i Registri Generali, cioè registri "visibili" a livello assembler (riferiti esplicitamente dalle istruzioni assembler), che svolgono un ruolo importante agli effetti delle ottimizzazioni introdotte dal compilatore. Ovviamente, come accade nel progetto di ogni unità di elaborazione, nell'unità P saranno presenti altri registri, visibili esclusivamente a livello firmware.

Il *formato* di una generica istruzione assembler è codificato come una stringa di bit nella quale si riconoscono *campi* corrispondenti alle seguenti informazioni (alcune possono essere implicite a seconda del Codice Operativo):

- Codice Operativo
- riferimenti (indirizzi) ad operandi e risultati
- informazioni su come individuare l'istruzione successiva.

Tale formato può essere codificato mediante una singola parola oppure mediante più parole consecutive (eventualmente in numero variabile a seconda della classe di istruzione); nel secondo caso, il Processore provvede a leggere dalla memoria il numero corrispondente di parole.

Ad alto livello, il *microprogramma del Processore* (interprete del linguaggio assembler, e quindi interprete del programma eseguibile) assume la forma seguente nel modello di Von Neumann:

while true

{ *chiamata dell'istruzione*: lettura dell'istruzione dalla memoria all'indirizzo IC;

decodifica dell'istruzione: riconoscimento dell'istruzione letta mediante il campo Codice Operativo;

esecuzione dell'istruzione: per ogni Codice Operativo verranno svolte azioni diverse, incluse eventuali letture e/o scritture di dati dalla memoria e/o dai Registri Generali;

aggiornamento del contatore istruzioni IC;

test di interruzioni: ascolto di eventuali segnalazioni da parte del sottosistema di I/O

}

Inizialmente, IC assume il valore dell'indirizzo della prima istruzione eseguibile.

2. I livelli assembler e firmware di un calcolatore general-purpose

Nel Cap. V della Dispensa è introdotto un set di istruzioni assembler di tipo *Risc*, e sono studiate regole di compilazione di programmi. L'esempio della sez. 1.1 delle Note Integrative sulla Parte Prima, per spiegare la differenza tra compilatore ed interprete, è un classico esercizio del Cap. V, che mostra come l'uso opportuno dei registri generali permetta di introdurre alcune importanti ottimizzazioni nel codice eseguibile.

Nello stesso Cap. V, e attraverso esercizi in queste note (sez. 4, 5), sono esaminate diverse alternative nella definizione del linguaggio assembler, sia di tipo Risc che Cisc.

Uno schema di Processore capace di interpretare il set di istruzioni assembler del Cap. V è studiato nel Cap. VII. Importante, in questo capitolo, è anche la definizione dei metodi per la valutazione delle prestazioni sulla base delle caratteristiche dell'architettura firmware e delle caratteristiche dei programmi.

3. Memoria virtuale e spazio di indirizzamento

Quello finora preso in considerazione è ancora uno schema semplificato dell'architettura, a causa di diversi aspetti legati alle caratteristiche di livelli superiori del sistema. In particolare, è importante il concetto di *memoria virtuale* e le conseguenze che tale concetto ha sulla compilazione e sull'architettura firmware. Nelle Dispense del corso, questo argomento è trattato per una parte minimale nel Cap. V, ed in modo più sostanziale nel Cap. VI, sez. 2, nel Cap. VII, sez. 1.1, e nel Cap. VIII, sez. 1.

Qui di seguito verrà dato un **sommario** per ora sufficiente a comprendere gli aspetti legati a questo importante argomento.

3.1. Spazio di indirizzamento fisico e spazio di indirizzamento logico

1. gli indirizzi generati da ogni programma non sono direttamente indirizzi di memoria principale (*indirizzi fisici*), bensì *indirizzi logici*, cioè indirizzi riferiti ad una astrazione della memoria del programma, detta *memoria virtuale*. Questa può essere vista come un array unidimensionale, con indici (indirizzi logici) a partire da zero fino al massimo necessario per rappresentare il programma o fino al massimo consentito dall'ampiezza dell'indirizzo logico in bit (ad esempio, per una macchina a 32 bit e con indirizzamento alla parola, la massima ampiezza della memoria virtuale di ogni programma è 4G parole). Ad esempio, le istruzioni del programma iniziano dall'indirizzo logico zero; dopo la zona della memoria virtuale occupata da istruzioni segue la zona occupata dai dati. L'insieme degli indirizzi logici di un programma è detto *spazio logico di indirizzamento* di quel programma;
2. il *codice eseguibile* del programma, generato dal compilatore, è quindi riferito alla memoria virtuale. Il Processore genera indirizzi logici sia per le istruzioni che per i dati;
3. il risultato della compilazione è un *file* binario ("file oggetto") che, come ogni file, viene conservato permanentemente in memoria secondaria e che, in tempi successivi qualsiasi, può venire utilizzato per eseguire il programma;
4. quando viene eseguito, il programma viene *allocato* in una zona della memoria principale, la cui ampiezza ed i cui indirizzi *non* coincidono (tranne casi particolari) con quelli della memoria virtuale del programma. In generale, per poter sfruttare convenientemente la "risorsa memoria principale", istante per istante ogni programma non è interamente allocato in memoria principale, né la parte allocata è memorizzata ad indirizzi contigui. La memoria principale viene *allocata dinamicamente* ad ogni programma, contando sul fatto che una copia intera, ed integra, del programma è sempre presente in memoria secondaria;
5. quando un programma viene allocato (o riallocato) in memoria principale, viene stabilita una corrispondenza tra gli indirizzi logici della parte allocata e gli indirizzi fisici in cui viene allocata. Questa funzione, detta *funzione di rilocazione o di traduzione dell'indirizzo*, viene di norma implementata come una tabella associata al programma (*Tabella di Rilocazione*), che fa parte essa stessa della memoria virtuale ed è allocata essa stessa in memoria principale;
6. la funzione viene aggiornata (dalla funzionalità del sistema operativo relativa alla gestione della memoria principale) ogni volta che l'allocazione del programma viene modificata. Ad esempio, in un certo istante un programma in esecuzione può avere bisogno di informazioni che non sono attualmente allocate in memoria principale. Queste verranno copiate da memoria secondaria a memoria principale. In generale, le nuove informazioni andranno a sostituirne altre "meno urgenti" (dello stesso programma o di altri programmi); se le informazioni sostituite sono state nel frattempo modificate (scritture in variabili), i loro valori verranno ricopiati

da memoria principale nelle posizioni corrispondenti della memoria secondaria. Questi spostamenti (*riallocazioni*) comportano opportune *modifiche della Tabella di Rilocazione* del programma (dei programmi). Si noti che il file eseguibile, presente in memoria secondaria, è sempre una *immagine affidabile* del programma;

7. la traduzione dell'indirizzo deve essere effettuata in modo molto efficiente (non più di un ciclo di clock del Processore), e quindi l'accesso alla Tabella di Rilocazione viene effettuata con opportune soluzioni hardware-firmware delegate ad una unità interposta tra il Processore e la memoria virtuale (*Memory Management Unit*, o *MMU*).

Quanto finora sintetizzato ha importanti conseguenze sulla compilazione e sull'esecuzione di programmi. Vediamone alcune.

3.2. Inizializzazione di variabili

Il compilatore, durante la traduzione di un programma, sceglie gli indirizzi logici di istruzioni e dati, e quindi *configura lo spazio di indirizzamento logico*, cioè lo spazio occupato dalla memoria virtuale di quel programma. Nel far questo, il compilatore può assegnare *valori iniziali* a variabili appartenente alla parte dati della memoria virtuale, oltre che valori iniziali a registri generali. Ad altre informazioni, che non hanno un valore iniziale, verrà comunque *riservato spazio* in memoria virtuale, senza prevederne un contenuto specifico. Il file eseguibile contiene quindi informazioni *inizializzate* (tutte le istruzioni, alcuni dati in memoria, alcuni registri generali) e *non inizializzate*.

Quando il (una parte del) programma verrà caricato in memoria per essere eseguito, automaticamente verranno inizializzate le locazioni di memoria principale nelle quali sono allocate istruzioni e dati (la parte delle istruzioni e dei dati allocata in memoria principale). Si veda la sezione successiva.

3.3. Caricamento ed esecuzione di un programma

Si veda Cap. V, sez. 3.4 e 4.10, Cap. VI, sez. 1.3.2 e sez. 2.3. Di seguito è riportata la sintesi di questo concetto:

- ad ogni programma (processo) è associato un *descrittore di processo* (*PCB*) contenente varie informazioni di utilità durante la vita del programma stesso, in particolare: *immagine dei valori dei registri generali, immagine del valore del contatore istruzioni, immagine della Tabella di Rilocazione,* ed altre informazioni;
- *il PCB fa parte della memoria virtuale del programma,* ed è quindi *inizializzato a tempo di compilazione*. Ciò significa che il file eseguibile contiene, nella parte PCB, i valori iniziali di IC e dei registri generali;
- quando viene lanciato il comando di esecuzione di un certo programma, il gestore della memoria principale (sistema operativo) provvede a individuare (se possibile) una parte della memoria principale in cui allocare una porzione del programma ("working set" del programma, vedi Cap. VIII, sez. 1.2). Tale porzione, che deve includere comunque il PCB, viene quindi copiata da memoria secondaria a memoria principale. Il PCB viene concatenato alla lista dei programmi eseguibili (Lista dei Processi Pronti, Cap. VI, sez. 1.3). Quando il Processore si rende disponibile, dal PCB primo in lista vengono copiate le immagini dei registri generali e del contatore istruzioni nei registri corrispondenti del Processore. Questo rende quindi possibile l'inizializzazione dei registri RG e IC mediante una sequenza di istruzioni assembler. Se l'ultima istruzione (START_PROCESS: vedi Cap. V, sez. 2.7) quella che provvede a caricare IC ed a fornire alla MMU le informazioni minime sulla nuova Tabella di Rilocazione, la prossima istruzione eseguita sarà la prima istruzione da cui deve iniziare (o riprendere) l'esecuzione del nuovo programma.

In termini di sistema operativo, la funzionalità ora vista corrisponde alla così detta *creazione* di un processo.

3.4. Trasferimenti di ingresso-uscita

Un altro con concetto molto importante è legato all'impatto che la memoria virtuale ha sull'implementazione dei *traferimenti di ingresso-uscita*.

L'implementazione, detta *Memory Mapped I/O,* che viene di seguito sintetizzata, si trova nel Cap. VI, sez. 3, Cap. VII, sez. 1.2, e Cap. IX, sez. 2.2.

Si consideri il seguente esempio: un programma che opera su un array A di *N* interi, assegna agli elementi di A i valori iniziali prelevandoli da un dispositivo di I/O (tastiera, disco, rete, …). Invece di introdurre nel programma complesse e intricate codice sequenze di istruzioni dipendenti dalla natura del dispositivo, si può più semplicemente ragionare come segue:

a) qualunque informazione presente nel sistema, incluso il sottosistema di I/O, viene vista come una informazione nella memoria virtuale del programma che deve utilizzare tale informazione. Questo è vero non solo per le informazioni che verranno allocate in memoria principale, ma anche per quelle *trasferite dai/verso i dispositivi di I/O*;

b) *ogni unità di I/O viene vista come se fosse un modulo di memoria fisica*: la memoria principale contiene quindi non solo il modulo (i moduli) della parte indicata come "memoria principale in senso stretto", ma anche tutte le memorie di I/O. In effetti, questa astrazione è molto vicina la vero: le unità di I/O contengono una loro memoria fisica (anche molto grande, ad esempio, qualche Kilo fino a qualche centinaio di Mega), realizzata come complesso di RAM, ROM, registri e interfacce;

c) il programma di cui sopra può essere scritto come segue:

```
for (i = 0; i < N; i++)
        A[i] = B[i]; …
```

dove B[N] è un array di interi appartenente alla memoria virtuale del programma, ma allocato nella memoria fisica di una qualche unità di I/O. Durante l'esecuzione dell'istruzione (LOAD) per trasferire il valore del generico B[i] in un registro generale, la traduzione dell'indirizzo logico di B[i] produrrà un indirizzo fisico che non è relativo alla memoria principale, bensì a quella specifica memoria di I/O. Sarà compito della MMU (si veda la fig. 2 del Cap. VII, sez. 1) instradare opportunamente la richiesta di accesso in memoria, attendere dall'unità di I/O il risultato dell'accesso e ritornarlo al Processore;

d) la *funzione di traduzione degli indirizzi*, relativi a dati allocati nello spazio di I/O (come l'array B nell'esempio), è prevista staticamente in base ad informazioni che il compilatore ha sulla configurazione del sottosistema di I/O ed in particolare sulle memorie di I/O.

4. Un esempio completo: compilazione, semantica delle istruzioni, memoria virtuale, caricamento

a) Compilare in assembler Risc (Cap. V) un programma così definito: trasforma un array A di N interi in un array B di N interi, con N = 100.000; note due costanti intere V1 e V2, con V2 > V1, il generico B[i] è uguale ad A[i] se V1 < A[i] < V2, altrimenti è uguale a V2 se A[i] ≥ V2 oppure è uguale a V1 se A[i] ≤ V1.

b) Scelta una istruzione appartenente ad ogni classe utilizzata nel programma compilato in *a)*, spiegarne il formato e la semantica, indicando esplicitamente i valori degli indirizzi dei registri e delle costanti utilizzate e le ampiezze dei campi.

c) Descrivere la memoria virtuale e lo spazio di indirizzamento del programma compilato in *a)*, spiegando quali locazioni della memoria virtuale sono inizializzate a tempo di compilazione. Per semplicità, si suppone che anche il valore dell'array A sia noto staticamente. [1]

d) Spiegare come si implementa il fatto che tutte quelle variabili o costanti, allocate in memoria o in registri generali, i cui valori iniziali sono fissati a tempo di compilazione, vengano effettivamente inizializzate a tali valori a tempo di esecuzione.

Risposta a)

Il programma ad alto livello può essere:

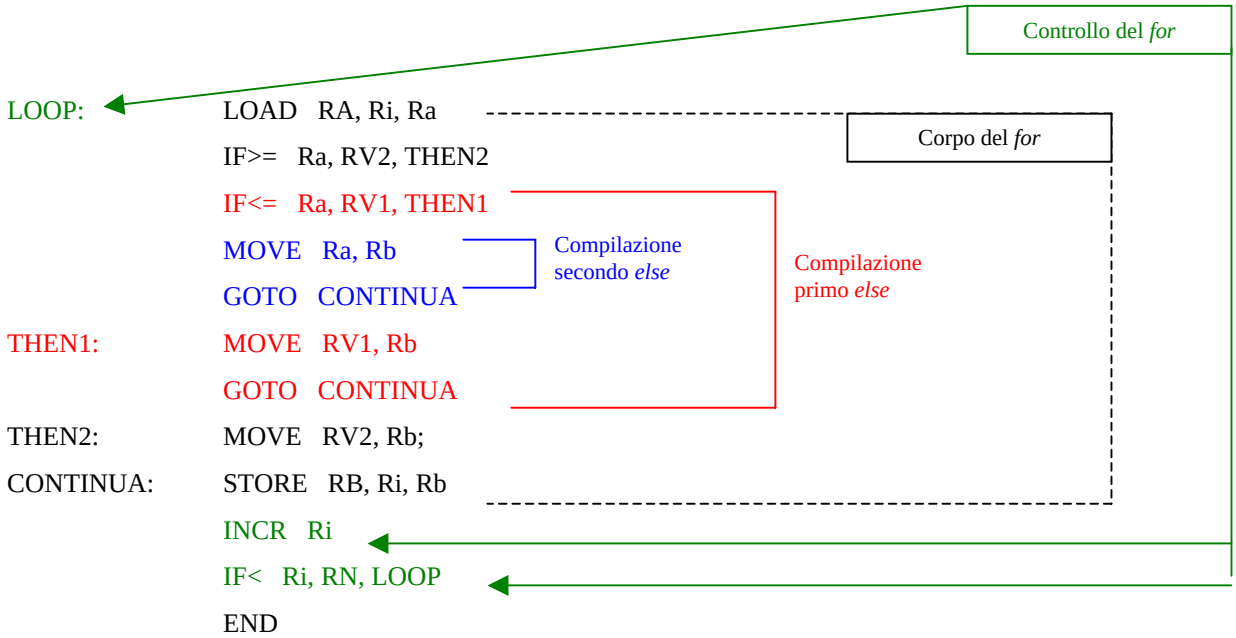
```
int N = 100000; int V1 = …; int V2 = …; int A[N], B[N];
{ for (i = 0; i < N; i ++)
        { if A[i] >= V2
                B[i] = V2;
        else
                if A[i] <= V1
                        B[i] = V1;
                else
                        B[i] = A[i];
        {
}
```

[1]. In un caso reale; A sarà un valore passato da un altro programma, oppure acquisito dall'esterno a tempo di esecuzione. Il codice per eseguire questo "aggancio" sarà oggetto della parte V del corso, e non verrà indicato esplicitamente nel codice compilato di questo esercizio.

Nel seguito faremo uso di nomi simbolici per gli indirizzi dei registri generali, ad esempio RA denota il registro generale (può essere R0) in cui viene allocata la base dell'array A.

Il compilatore:

- alloca spazio di memoria virtuale agli array A e B, determinando quindi i rispettivi indirizzi base *base_A* e *base_B*;
- alloca e inizializza i registri generali RA, RB, RN, RV1 e RV2 rispettivamente alle costanti base_A, base_B, N, V1 e V2, ed il registro generale Ri, inizializzato a zero, alla variabile *i*. Tali registri saranno, in ordine, quelli da R0 a R5;
- alloca i registri temporanei, quindi non inizializzati, Ra e Rb, corrispondenti rispettivamente a R6 e R7;
- applicando le regole di traduzione dei comandi *if-then-else* e *for* (quest'ultimo implementato come *do_while*, in quanto il compilatore conosce che N > 0), genera il seguente codice:



Risposta b)

Istruzione di trasferimento: LOAD RA, Ri, Ra

Campi del formato su singola parola:

- codice operativo per "LOAD" con i valori di entrambi gli addendi (base e indice), per il calcolo dell'indirizzo, in registri generali (8 bit),
- indirizzo *0* di registro generale (6 bit),
- indirizzo *5* di registro generale (6),
- indirizzo *6* di registro generale (6),
- non specificati (6).

Semantica: il contenuto della locazione di memoria virtuale, il cui indirizzo è dato dalla somma dei contenuti (indirizzi base e indice) dei registri generali di indirizzo *0* e *5*, viene scritto nel registro generale di indirizzo *6*; il contatore istruzioni viene incrementato di 1:

$$MV[\ RG[0] + RG[5]\ ] \rightarrow RG[6]\ ,\ IC + 1 \rightarrow IC.$$

Istruzione di trasferimento: STORE RB, Ri, Rb

Campi del formato su singola parola:

- codice operativo per "STORE" con i valori di entrambi gli addendi (base e indice), per il calcolo dell'indirizzo, in registri generali (8 bit),

- indirizzo *1* di registro generale (6 bit),
- indirizzo *5* di registro generale (6),
- indirizzo *7* di registro generale (6),
- non specificati (6).

Semantica: il contenuto del registro generale di indirizzo *7* viene scritto nella locazione di memoria virtuale, il cui indirizzo è dato dalla somma dei contenuti dei registri generali di indirizzo *1* e *5*, il contatore istruzioni viene incrementato di 1:

$RG[7] \rightarrow MV[\ RG[1] + RG[5]\ ]\ ,\ IC + 1 \rightarrow IC.$

Istruzione di salto condizionato: IF< Ri, RN, LOOP

Campi del formato su singola parola:

- codice operativo per "IF<" con indirizzo di salto relativo a IC (8 bit),
- indirizzo *5* di registro generale (6 bit),
- indirizzo *2* di registro generale (6),
- offset = -10, rappresentato su 12 bit.

Semantica: se il contenuto del registro generale di indirizzo *5* è minore del contenuto del registro generale di indirizzo *2*, allora al contatore istruzioni viene sommata la costante -10, altrimenti il contatore istruzioni viene incrementato di 1:

if **(** $RG[5] < RG[2]$) ***then*** $IC - 10 \rightarrow IC$ ***else*** $IC + 1 \rightarrow IC.$

Istruzione di salto incondizionato: la prima delle due istruzioni GOTO CONTINUA

Campi del formato su singola parola:

- codice operativo per "GOTO" con indirizzo di salto relativo a IC (8 bit),
- offset = +4, rappresentato su 24 bit.

Semantica: al contatore istruzioni viene sommata la costante +4:

$IC + 4 \rightarrow IC.$

Istruzione operativa: INCR Ri

Campi del formato su singola parola:

- codice operativo per "INCR" con operando contenuto in registro generale (8 bit),
- indirizzo *5* di registro generale RG (6 bit),
- non specificati (18).

Semantica: l'incremento del contenuto del registro generale di indirizzo *5* viene scritto nel registro stesso; il contatore istruzioni viene incrementato di 1:

$RG[5] + 1 \rightarrow RG[5]\ ,\ IC + 1 \rightarrow IC.$

Istruzione operativa: MOVE RV2, Rb

Campi del formato su singola parola:

- codice operativo per "MOVE" con operandi contenuti in registri generali (8 bit),
- indirizzo *4* di registro generale RG (6 bit),
- indirizzo *7* di registro generale RG (6),
- non specificati (12).

Semantica: il contenuto del registro generale di indirizzo *4* viene scritto nel registro generale di indirizzo *7*; il contatore istruzioni viene incrementato di 1:

$RG[4] \rightarrow RG[7]\ ,\ IC + 1 \rightarrow IC.$

Istruzione speciale: END

Campi del formato su singola parola:

- codice operativo per “END” (8 bit),
- non specificati (24 bit).

Semantica: chiamata della funzionalità di nucleo del sistema operativo per gestire la terminazione del processo corrispondente al programma.

Risposta c)

Il compilatore può allocare spazio per gli array A e B nella parte iniziale della memoria virtuale MV.

Segue la parte allocata alle istruzioni del programma, e quindi lo spazio per il Descrittore del Programma (PCB).

Verrà poi allocata memoria virtuale per funzionalità di nucleo di sistema operativo (non trattato in questo esercizio).

Nel seguito, per ogni oggetto allocato in MV verranno indicati:

intervallo di indirizzi logici dello spazio di indirizzamento **:** *oggetto allocato* **;** *indicazione della sua eventuale inizializzazione statica.*

MV ::

- 0 .. 99.999 : array A ; inizializzato /vedi nota a fondo pagina 1/
- 100.000 .. 199.999 : array B ; non inizializzato
- 200.000 .. 200.011 : istruzioni del programma ; inizializzato
- 200.012 .. 200.267 : struttura dati PCB /nota: si suppone che siano necessarie complessivamente 256 parole per tutto il PCB; per il momento interessa specificare solo la parte “immagine di RG” e “immagine di IC”, che complessivamente occupano 65 parole/ ; nell’immagine di RG sono inizializzate le prime 8 parole, le altre sono non specificate; l’immagine di IC è inizializzata al valore 200.000; /anche alcune parole della rimanente parte del PCB saranno inizializzate/
- 200.267 ….. : altre informazioni di nucleo del sistema operativo /in parte inizializzate/.

Risposta d)

Quando viene lanciato il comando di esecuzione del programma, una opportuna funzionalità del sistema operativo (gestore della memoria principale) provvede a individuare (se esiste momentaneamente) una parte della memoria principale in cui allocare almeno una porzione significativa del programma. Tale porzione, che deve includere necessariamente il PCB, viene quindi copiata da memoria secondaria (file eseguibile in cui è codificata la memoria virtuale del programma) in memoria principale. Il PCB viene concatenato ad una lista di programmi eseguibili.

Inizializzazione di variabili/costanti in memoria: quelle che sono inizializzate nel file eseguibile, verranno automaticamente copiate in memoria principale all’atto del loro trasferimento. (Quelle che verranno modificate durante l’esecuzione saranno poi ricopiate nel file eseguibile alla fine dell’esecuzione, o quando verranno deallocate).

Inizializzazione di variabili/costanti in RB e inizializzazione di IC: quando il Processore si rende disponibile, dal PCB primo in lista vengono copiate le immagini dei registri generali e del contatore istruzioni nei registri corrispondenti del Processore: *fase di commutazione di contesto* (preceduta dall’operazione “inversa” per il programma precedentemente in esecuzione). Questo rende quindi possibile l’inizializzazione dei registri RG e IC mediante una normale sequenza di istruzioni assembler (facenti parte del nucleo del sistema operativo). La prossima istruzione eseguita, dopo questa sequenza, sarà quindi proprio la prima istruzione da cui deve iniziare (o riprendere) l’esecuzione del programma con il contenuto corretto di RG.

Si rifletta sul fatto che il Processore, a livello firmware, esegue permanentemente, ed esclusivamente, l’interprete del linguaggio assembler, eseguendo istante per istante l’istruzione il cui indirizzo logico è contento in IC ed operando sui contenuti attuali di RG. Il Processore non è cosciente di eseguire questo o quel programma: la corretta esecuzione dei programmi è resa possibile proprio dal meccanismo dell’inizializzazione delle variabili/costanti e della commutazione di contesto, quest’ultimo implementato esso stesso come sequenza di istruzioni assembler.

5. Esercizi sulla macchina assembler e spazi di indirizzamento

1) Sia data una macchina assembler con 32 registri generali RG [0 .. 31]; la parola è di 32 bit; gli indirizzi logici sono di 32 bit; si indichi con MV la memoria virtuale di un programma. Spiegare il significato delle seguenti scritture:

RG [5] ;

MV [RG [5]] ;

MV [RG [5] + RG [13]] ;

MV [43971] ;

MV [MV [43971]];

MV [MV [RG [5] + RG [13]]];

MV [IC + 3825]

2) Compilare in assembler Risc un programma che, dati due array bidimensionali di interi, A[N][N] e B[N][N], con N = 4K, restituisce, in una data locazione di memoria virtuale, il numero di componenti di A e B, con gli stessi indici, che hanno lo stesso valore.

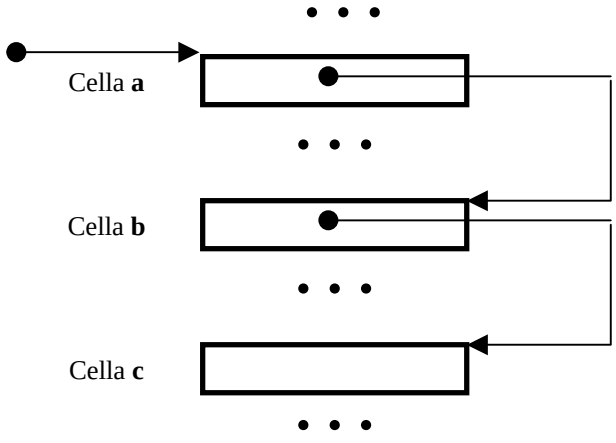
Mostrare la configurazione della memoria virtuale e dello spazio logico di indirizzamento.

3) Compilare un programma che esegue la funzione detta "Sum-prefix" [2]: dato un array unidimensionale A di N interi, restituisce un array unidimensionale B di N interi così definito:

$$\forall\, i = 0..N-1 : B[i] = \sum_{j=0}^{i} A[j]$$

Mostrare la configurazione della memoria virtuale e dello spazio logico di indirizzamento.

4) Si consideri la seguente configurazione di una parte della memoria virtuale di un programma, dove i rettangoli denotano locazioni e le frecce indirizzi logici usati come puntatori:



Assumendo come unica informazione nota a priori il fatto che l'indirizzo della locazione indicata con "Cella a" sia contenuto nel registro generale di indirizzo 7:

a) scrivere una sequenza di istruzioni Risc che permetta di decrementare di uno il contenuto della locazione indicata con "Cella c";

[2] In generale, ⊗-Prefix, dove ⊗ è una qualunque operazione associativa, è un operatore del secondo ordine, B = Prefix(A, ⊗), che restituisce in B tutti i possibili "prefissi di A secondo l'operatore ⊗": $B_0 = A_0$, $B_1 = A_0 \otimes A_1$, …, $B_{N-1} = A_0 \otimes A_1 \otimes \ldots \otimes A_{N-1}$.

b) scrivere una sequenza di istruzioni Risc che permetta di effettuare un salto all'istruzione il cui indirizzo è dato dal contenuto della locazione indicata con "Cella c".

5) Una tabella, realizzata mediante un vettore di N posizioni, contiene in ogni posizione l'indirizzo della prima istruzione di un programma assembler contraddistinto da un intero non negativo ($prog_0, prog_1, \ldots prog_{N-1}$).

Scrivere una sequenza di istruzioni assembler Risc che, dati attraverso due registri generali l'indirizzo base della tabella ed un valore intero *j*, permette di saltare alla prima istruzione del programma $prog_j$.

6) Compilare in assembler Risc i seguenti programmi che operano su una lista "linkata" (con puntatori) unidirezionale, con elementi di tipo intero:

- creazione di una lista vuota;
- inserzione in testa;
- estrazione dalla testa;
- ricerca di un valore.

Mostrare la configurazione della memoria virtuale e dello spazio logico di indirizzamento.

7) Trasformare i programmi visti negli esercizi precedenti in *procedure*, scrivendo il codice che il compilatore inserisce nei programmi chiamanti per invocare tali procedure e per passare loro i parametri nella maniera ritenuta più conveniente.

8) Si consideri il seguente programma compilato in assembler Risc:

```
LOOP:       LOAD  R1, R2, R3
            LOAD  R3, R0, R4
            IF<=0  R4, CONTINUA
            ADD  R5, R4, R5
CONTINUA:   INCR  R2
            IF<  R2, R6, LOOP
            STORE  R7, R0, R5
            GOTO  R8
```

I registri generali di indirizzo 0, 2 e 5 sono inizializzati a 0, quello di indirizzo 1 a 100, quello di indirizzo 6 a 3000, quello di indirizzo 7 a 10000.

a) Spiegare quale funzionalità viene elaborata dal programma.

b) Dire *quanti* sono gli indirizzi che possono essere generati a tempo di esecuzione dal processore, senza considerare gli indirizzi delle informazioni utilizzate per funzionalità di sistema operativo. È possibile dire *quali* indirizzi sono generati a tempo di esecuzione dal processore ?

c) Spiegare come i registri generali di indirizzo 0, 1, 2, 5, 6, 7 vengono effettivamente inizializzati al valori indicati sopra.

6. Esercizi sull'architettura del processore

1) Compilare in assembler Risc (Cap. V) un programma che conta il numero di elementi non negativi in un array di N interi.

Si suppone che questo solo programma sia ritenuto caratterizzante un certo campo di applicazione. La probabilità che un numero sia non negativo è ritenuta uguale a 0,75. Valutarne il tempo di completamento, il tempo medio di elaborazione e la performance (*per quel campo di applicazione*) per un calcolatore avente processore con ciclo di clock τ = 0,5 nsec, memoria principale con ciclo di clock di 50τ e latenza di trasmissione dei collegamenti inter-chip di 10τ.

2) Per un calcolatore con le caratteristiche indicate nell'Esercizio 1, scrivere il microprogramma di esecuzione e valutare il tempo medio di elaborazione delle seguenti istruzioni Risc (Cap. V):

IF > Ri, Rj, offset

MUL Ri, Rj, Rk il risultato è una doppia parola da scrivere nei due registri generali di indirizzo *k* e *k+ 1*; usare l'algoritmo di moltiplicazione del Cap. III, sez. 3.8, Es. 1.

3) Le seguenti istruzioni *non* appartengono al linguaggio assembler del Cap. V:

a) IFM=0 Ri, Rj, Rk significato: *if MEM [RG[i]] = MEM [RG[j]] then GOTO [RG[k]]*

b) IFMM =0 Ri, Rj, Rk significato: *come sopra ma con GOTO MEM[RG[k]]*

c) ADDV Ri, Rj, Rk, Rd significato:

d) *MEM [RG[k] + RG[d]] = MEM [RG[i] + RG[d]] + MEM [RG[j] + RG[d]]*

Supponendo di aggiungere queste istruzioni al linguaggio assembler del Cap. V (supponendo che ci siano altrettanti codici operativi disponibili), darne il formato, scriverne il microprogramma di esecuzione e valutarne il tempo medio di elaborazione per il calcolatore con le caratteristiche dell'Esercizio 1.

Usando anche l'istruzione ADDV, compilare un programma che esegua la somma di due array A e B, di N interi, producendo un array C, e valutarne il tempo di completamento. Confrontare il risultato della compilazione, ed il tempo di completamento, con quelli ottenibili usando l'assembler Risc puro (Cap. V).

4) Esercizi 2, 5, sez. 5 Cap. VII.

5) Usando la corretta terminologia, rispondere alle seguenti domande.

- Spiegare in cosa consiste la performance di un calcolatore e come viene valutata.
- Siano dati due calcolatori C1, C2, aventi lo stesso ciclo di clock e la stessa gerarchia di memoria. Il set di istruzioni di C1 è quello del Cap. V, il set di istruzioni di C2 è quello del Cap. V arricchito da quattro istruzioni per operare su numeri reali. Spiegare quale dei due calcolatori ha performance maggiore.

6) Valutare il tempo di elaborazione del trattamento eccezioni, a partire dall'istante in cui l'eccezione viene rilevata all'istante in cui la "routine assembler di interfaccia tra fase firmware e Handler" chiama il programma Handler.

7) La seguente istruzione, tutt'altro che Risc, *non* appartiene al linguaggio assembler del Cap. V. Si tratta di una istruzione di addizione *a formato variabile*: le informazioni da utilizzare, e la stessa lunghezza dell'istruzione (in parole), dipendono dal valore di un campo *MODE* contenuto nella prima parola. La parola è di 32 bit. I registri generali sono 64. I campi sono i seguenti:

- codice operativo di 8 bit;
- *MODE* di 2 bit;
- *i, j k:* indirizzi di registri generali;
- **se MODE = 00**, il significato è: *RG[k] = RG[i] + RG[j];*
- 4 bit inutilizzati;
- *BASE* di 32 bit; questo campo esiste solo **se MODE = 01**, **10, 11**;
- se MODE = 01, il significato è: *RG[k] = MEM [BASE + RG[i]] + R[j];*
- se MODE = 11, il significato è: *MEM[RG[k]]= MEM [BASE + RG[i]] + R[j];*
- *INDICE* di 32 bit; questo campo esiste solo **se MODE = 10**; in questo caso, il significato è: *RG[k] = MEM [BASE + INDICE] + R[j]*.

Darne il formato e scrivere il microprogramma che la interpreta completamente.