Architettura degli Elaboratori, 2006-07

Seconda Esercitazione di Verifica Intermedia

Consegna: lezione di venerdì 17 novembre, ore 11

L'elaborato, da presentare in una forma leggibile agevolmente, deve contenere spiegazioni chiare ed esaurienti, utilizzando la corretta terminologia ed i concetti del corso.
Insieme a nome e cognome indicare l'anno di corso di ogni studente.

Domanda 1

a) Compilare in assembler Risc un programma così definito:

- opera sugli array di interi A[N], B[N], C[N], con $N = 2^{20}$;
- $\forall i = 0 .. N - 1 : C[i] = max (C[i], B[j])$, dove $j = (abs (A[i])) \bmod N$

Il programma va implementato come procedura i cui parametri di ingresso (A, B, C) e di uscita (C) sono passati per indirizzo via locazioni di memoria.

Spiegare come sono state applicate le regole di compilazione e quali modi di indirizzamento sono stati utilizzati nella compilazione.

b) Descrivere la memoria virtuale e lo spazio di indirizzamento del programma, spiegando quali locazioni della memoria virtuale sono inizializzate a tempo di compilazione. Si supponga che il descrittore di processo occupi 128 parole e che le altre informazioni di sistema operativo "collegate" al programma occupino complessivamente 8K parole.

c) Spiegare qual è il significato e l'implementazione della frase "inizializzazione di un registro generale o di una locazione di memoria a tempo di compilazione".

Domanda 2

a) Si supponga che il programma della Domanda 1 sia caratterizzante di un certo campo di applicazione.

Valutare il tempo di completamento del programma e la performance per un calcolatore avente processore con clock a 4 GHz, memoria principale con clock a 100 MHz e latenza di trasmissione dei collegamenti inter-chip uguale a 20 volte il ciclo di clock del processore.

Si assumano trascurabili la probabilità di non trovare una informazione del programma in memoria principale e la probabilità che una unità di I/O invii una interruzione.

b) Supponiamo che al set di istruzioni Risc siano aggiunte istruzioni con operandi in memoria sia per la classe delle aritmetico-logiche che per quelle di salto condizionato.

Dare il formato di tali istruzioni e scriverne l'interprete con riferimento alle operazioni aritmetiche ed ai predicati che sono utilizzati nel programma della Domanda 1.

Valutare l'eventuale vantaggio che l'uso di tali istruzioni comporterebbe sul tempo di completamento e sulla performance del programma della Domanda 1.

Domanda 3

Dire se le seguenti affermazioni sono vere o false o vere sotto determinate condizioni, spiegando le risposte:

1. Un programma assembler A viene interpretato da un proprio microprogramma, diverso da quello che interpreta un programma assembler B.
2. In un elaboratore general purpose con capacità massima di memoria principale di 1G parole, gli indirizzi generati in istruzioni di Load e Store sono espressi come numeri naturali di 30 bit.
3. Lo spazio di indirizzamento di un programma è l'insieme di tutti i possibili indirizzi logici che possono essere generati dal programma in esecuzione.
4. La MMU di un calcolatore implementa una funzionalità di sistema operativo.
5. Sostituendo la memoria principale e/o la MMU è necessario modificare i microprogrammi del processore.
6. Un certo algoritmo che opera su array è implementato in due modi diversi: *a)* a livello assembler per un calcolatore general-purpose, *b)* a livello firmware in una unità di elaborazione U, avente lo stesso ciclo di clock del processore del calcolatore di cui al punto *a)* e collegata ad una memoria esterna avente le stesse prestazioni della memoria principale del calcolatore di cui al punto *a)*. L'affermazione è : il tempo di completamento nel caso *a)* è maggiore di quello del caso *b)* e la differenza è proporzionale al tempo di accesso in memoria moltiplicato per il numero delle istruzioni eseguite dal programma *a)*.

Domanda 1

a) I parametri d'ingresso della procedura sono gli *indirizzi* pA, pB, pC di tre locazioni di memoria virtuale che contengono altrettanti *puntatori* alle basi degli array A, B, C rispettivamente, come mostrato in figura. Si tratta di *indirizzamento indiretto via memoria*.

Il parametro di uscita della procedura è l'indirizzo pC.

I tre parametri sono i contenuti di altrettanti registri generali i cui indirizzi saranno indicati con **RpA**, **RpB**, **RpC**.

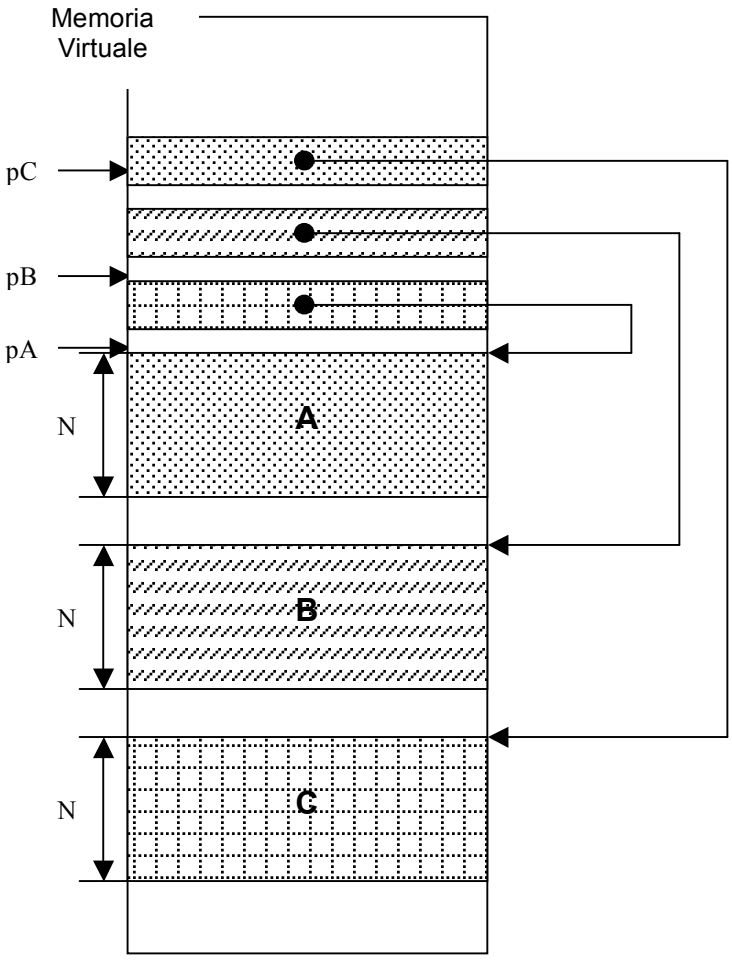
Sia **Rret** l'indirizzo del registro generale in cui è contenuto l'indirizzo di ritorno dalla procedura.

Sia **Rproc** l'indirizzo del registro generale in cui è inserito (all'atto della CALL) l'indirizzo della prima istruzione della procedura.

Supponiamo che la procedura sia **compilata assieme al programma principale**. Oltre ai cinque registri suddetti, il compilatore alloca *per la procedura* i registri generali :

- di indirizzi **RA**, **RB**, **RC**, contenenti gli indirizzi base dei tre array;
- di indirizzo **Ri**, **Rj**, contenenti gli indici i, j;
- di indirizzo **RN**, contenente il valore N;
- di indirizzi **Ra**, **Rb**, **Rc**, contenenti valori temporanei.

I registri Rproc e RN sono *inizializzati* a tempo di compilazione.



Nel caso che la procedura non sia compilata assieme al programma principale, bensì che sia una **libreria** preesistente (quindi, già in binario), il compilatore del programma principale sia sarebbe limitato a collegarla così com'è, prendendo atto dei registri da essa utilizzati; nel caso alcuni di questi fossero utilizzati dal programma principale prima della chiamata, i loro valori sarebbero stati salvati in locazioni di memoria virtuale (PCB) e ripristinati in seguito.

In questa sede assumeremo che "il programma consista solo della procedura", cioè che il programma principale contenga solo il passaggio dei parametri e la chiamata della procedura, terminando immediatamente al ritorno dalla stessa. Assumendo che i valori dei tre parametri siano contenuti (*inizializzati,* nel caso semplificato che stiamo considerando) nei registri **Rparam1**, **Rparam2**, **Rparam3** :

```
MOV   Rparam1, RpA
MOV   Rparam2, RpB
MOV   Rparam3, RpC
CALL   Rproc, Rret
END
```

Se, al ritorno dalla procedura, l'array C venisse riutilizzato, esso sarebbe disponibile usando l'indirizzamento indiretto via MEM[Rparam3], oppure l'indirizzamento indiretto via MEM[RpC].

La procedura comincia con l'assegnamento dei valori iniziali a RA, RB, RC, Ri:

```
LOAD   RpA, 0, RA
LOAD   RpB, 0, RB
LOAD   RpC, 0, RC
CLEAR  Ri
```

Le tre LOAD servono ad implementare l'indirizzamento indiretto via memoria (puntatori), che non è primitivo nella macchina assembler data. L'istruzione CLEAR inizializza dinamicamente l'indice *i*.

Da questo momento in poi, verranno sempre utilizzate le modalità di indirizzamento primitive della macchina assembler data: base + indice via registro per i dati riferiti dalle LOAD e STORE, relativo a IC nelle istruzioni di IF, indiretto via registro per il "goto calcolato" che implementa il ritorno da procedura.

La struttura del resto del programma compilato è:

```
< compilazione del for >
GOTO Rret
```

Compiliamo il loop corrispondente a $\forall\, i = 0 \,..\, N-1 : C[i] = max\,(C[i], B[j])$, dove $j = (\, abs\,(A[i])\,)\ mod\ N$, secondo un costrutto del tipo:

```
for (i = 0; i < N, i++)
        < corpo del for >
```

Essendo il loop eseguito N volte, con N > 0, il *for* viene compilato con le regola *do-while*:

```
LOOP:   < corpo del for >
        INCR   Ri
        IF<    Ri, RN, LOOP
```

Il corpo del *for* inizia con il calcolo dell'indice *j*:

```
if A[i] < 0
        A[i] = – A[i];
j = A[i] % N;
```

Per la sua compilazione, viene applicata la regola di compilazione *if-then*. Il calcolo del modulo viene effettuato con l'istruzione primitiva MOD:

```
LOOP:   LOAD   RA, Ri, Ra
        IF>=0  Ra, CONT1
        SUB    0, Ra, Ra
CONT1:  MOD    Ra, RN, Rj
```

Segue il calcolo del massimo:

```
if C[i] < B[j]
        C[i] = B[j]
```

compilato usando ancora la regola *if-then* :

```
        LOAD   RC, Ri, Rc
        LOAD   RB, Rj, Rb
        IF>=   Rc, Rb, CONT2
        STORE  RC, Ri, Rb
CONT2:  INCR   Ri
        IF<    Ri, RN, LOOP
```

Per composizione, si ottiene la compilazione completa della procedura:

```
        LOAD   RpA, 0, RA
        LOAD   RpB, 0, RB
        LOAD   RpC, 0, RC
        CLEAR  Ri
LOOP:   LOAD   RA, Ri, Ra
        IF>=0  Ra, CONT1
        SUB    0, Ra, Ra
CONT1:  MOD    Ra, RN, Rj
        LOAD   RC, Ri, Rc
        LOAD   RB, Rj, Rb
        IF>=   Rc, Rb, CONT2
        STORE  RC, Ri, Rb
CONT2:  INCR   Ri
        IF<    Ri, RN, LOOP
        GOTO   Rret
```

b) La memoria virtuale del programma contiene le seguenti locazioni, per le quali sono indicati gli indirizzi logici e se sono inizializzate o meno a tempo di compilazione:

- istruzioni: programma principale (5) + procedura (15): 20 locazioni (indirizzi logici da 0 a 19) inizializzate al codice binario delle istruzioni;
- puntatori agli array A, B, C: 3 locazioni (indirizzi logici da 20 a 22), inizializzate
- array A, B, C: 3Mega parole (indirizzi logici da 23 a 22+3M), inizializzate
- descrittore di processo (PCB): 128 locazioni (indirizzi da 23+3M a 150+3M), in parte inizializzate, in particolare le immagini dei registri RpA, RpB, RpC, RN e di IC, e il puntatore alla Tabella di Rilocazione;
- altre informazioni di sistema operativo: 8K locazioni (da 151+3M a 150+3M+8K), in parte inizializzate, in particolare la Tabella di Rilocazione (entrate inizializzate a "null" per quanto riguarda gli indirizzi fisici) ed altre informazioni che saranno viste in una parte successiva del corso.

c) Tutta la memoria virtuale ora descritta forma il file binario eseguibile (file oggetto), nel quale alcune informazioni hanno un valore inizializzato e le altre hanno un valore non significativo.

All'atto del caricamento in memoria (*creazione del processo*), le informazioni in memoria virtuale verranno copiate in opportune locazioni della memoria principale M (eventualmente, solo una parte di esse, tra cui almeno il PCB e le prime istruzioni), e dunque alcune locazioni di M assumeranno il desiderato valore iniziale.

Quando il processo passerà in *stato di esecuzione*, le immagini dei registri generali e di IC verranno copiate nei registri del processore, provocando così la loro inizializzazione.

Domanda 2

La CPU (processore, MMU) ha ciclo di clock

$$\tau = 10^{-9}/4 = 0{,}25 \text{ nsec}$$

l'unità M di memoria principale:

$$\tau_M = 40\ \tau$$

La latenza di trasmissione dei collegamenti tra CPU e M vale:

$$T_{tr} = 20\ \tau$$

Calcoliamo il tempo di completamento T_c del programma. Rigorosamente, questo tempo deve comprendere la fase di passaggio dei parametri, di chiamata della procedura e di END del programma principale, più il tempo di completamento della procedura. In pratica, dato l'elevato valore di N, T_c viene valutato come il tempo di completamento del solo *loop* della procedura.

Essendoci dei salti all'interno del loop, distinguiamo i possibili rami che possono essere percorsi dall'esecuzione del programma all'interno di una generica iterazione:

1) se A[i] ≥ 0, C[i] ≥ B[j]: **load; if; mod; load; load; if; incr; if**	8 istruzioni per iterazione
2) se A[i] < 0, C[i] ≥ B[j]: **load; if**; sub; **mod; load; load; if; incr; if**	9 istruzioni per iterazione
3) se A[i] ≥ 0, C[i] < B[j]: **load; if; mod; load; load; if;** store; **incr; if**	9 istruzioni per iterazione
4) se A[i] < 0, C[i] < B[j]: **load; if;** sub; **mod; load; load; if;** store; **incr; if**	10 istruzioni per iterazione

Non disponendo di informazioni sulle probabilità dei predicati, dobbiamo assumere che i quattro rami siano equiprobabili. Tenendo conto che

- le 8 istruzioni scritte in grassetto sono eseguite con probabilità 1,
- l'istruzione SUB è eseguita con probabilità uguale a 2/ 4 = 0,5,
- l'istruzione STORE è eseguita con probabilità uguale a 2/ 4 = 0,5,

le istruzioni eseguite in media per ogni iterazione sono 8 + 0,5 + 0,5 = 9.

Il tempo di completamento si calcola come:

$$T_c = N\ (9\ T_{ch} + 3{,}5\ T_{ex-LD} + 3\ T_{ex-IF} + T_{ex-MOD} + 1{,}5\ T_{ex-INCR})$$

essendo l'istruzione SUB appartenente alla classe delle aritmetiche-logiche corte come la INCR, e la STORE alla stessa classe della LOAD.

Sostituendo le espressioni dei tempi medi di chiamata-decodifica:

$$T_{ch} = 2\tau + t_a$$

e di esecuzione:

$$T_{ex-LD} = 2\tau + t_a$$
$$T_{ex-IF} = 2\tau$$
$$T_{ex-MOD} \cong 50\tau$$
$$T_{ex-INCR} = \tau$$

(l'istruzione MOD, di classe aritmetico-logiche lunghe, è stata valutata come una MUL), si ottiene:

$$T_c = N\ (82{,}5\ \tau + 12{,}5\ t_a)$$

Il tempo di accesso in memoria principale, come "visto" dal processore, è dato da:

$$t_a = \tau_M + 2(\tau + T_{tr}) = 82\tau$$

da cui:

$$T_c = 1107{,}5\,\tau\ N$$

Questa valutazione è valida sotto le ipotesi del testo circa la probabilità trascurabile di non trovare informazioni in memoria principale e che si verifichino interruzioni.

Numericamente (ricordando che 1 Mega = 1024 x 1024):

$$T_c \cong 0{,}21\,\text{sec}$$

Per il campo di applicazione caratterizzato da questo programma, il tempo medio di elaborazione si può calcolare partendo dal tempo di completamento; indicando con $k = 9N$ il numero medio di istruzioni elaborate, si ha:

$$T = \frac{T_c}{k} = \frac{1107{,}5\,\tau\ N}{9\ N} \cong 123{,}1\,\tau \cong 30{,}8\,n\,\text{sec}$$

da cui la *performance*:

$$\wp = \frac{1}{T} = 32{,}5\ MIPS$$

Lo stesso risultato si sarebbe ottenuto partendo dal tempo medio di elaborazione e ricavando da questo il tempo di completamento. Per il campo di applicazione caratterizzato da questo programma, il MIX di probabilità è dato da:

$$p_{LD-ST} = \frac{\frac{3}{8}+\frac{3}{9}+\frac{4}{9}+\frac{4}{10}}{4} \quad p_{IF} = \frac{\frac{3}{8}+\frac{3}{9}+\frac{3}{9}+\frac{3}{10}}{4} \quad p_{MOD} = \frac{\frac{1}{8}+\frac{1}{9}+\frac{1}{9}+\frac{1}{10}}{4} \quad p_{INCR-SUB} = \frac{\frac{1}{8}+\frac{2}{9}+\frac{1}{9}+\frac{2}{10}}{4}$$

e tutte le altre probabilità uguali a zero.

b) Supponiamo che al set di istruzioni Risc siano aggiunte istruzioni con operandi in memoria sia per la classe delle aritmetico-logiche che per quelle di salto condizionato.

Con riferimento ai predicati di interesse del programma, le nuove istruzioni di salto condizionato potrebbero essere:

IFM1>=0 Rbase, Rindice, LABEL1

IFM2>= Rindirizzo1, Rindirizzo2, LABEL2

Si noti che, se vogliamo mantenere la caratteristica che le istruzioni siano rappresentate *su singola parola*, la IFM2, con predicato diadico, non potrebbe prevedere quattro indirizzi di registri (due per le basi e due per gli due indici); dovendo prevedere due indirizzi in altrettanti registri, l'istruzione deve, dunque, essere preceduta da altre istruzioni (registro-registro) per calcolare tali indirizzi.

Invece, la IFM1, con predicato monodico può essere rappresentata su singola parola anche con indirizzamento base+indice.

Rispetto ai microprogrammi di istruzioni di salto condizionato con operandi in registri generali, i microprogrammi delle corrispondenti istruzioni con operandi in memoria contengono anche le sequenze di microistruzioni per la lettura degli operandi stessi.

Nel caso della IFM2>=, il microprogramma è:

ifm0. RG[IR.Rindirizzo1] → IND, 'read' → OP, set RDYOUT, **ifm1**

ifm1. (RDYIN, or(ESITO) = 0-) nop, ifm1;

(= 11) …., tratt_ecc ;

(= 10) reset RDYIN, DATAIN → A, RG[IR.Rindirizzo2] → IND, 'read' → OP, set RDYOUT, **ifm2**

ifm2. (RDYIN, or(ESITO) = 0-) nop, ifm2;

(= 11) …., tratt_ecc ;

(= 10) reset RDYIN, segno (A – DATAIN) → S, **ifm3**

ifm3. (S, INT = 0 0) IC + 1 → IC, ch0 ;

(S, INT = 0 1) IC + 1 → IC, tratt_int ;

(S, INT = 1 0) IC + IR.OFFsET → IC, ch0 ;

(Z, INT = 1 1) IC + IR.OFFSET → IC, tratt_int

Si osservi che *ogni accesso in memoria a dati* costa un tempo di elaborazione pari a $(\tau + t_a)$. Quindi

$$T_{ex\text{-}IFM1} = T_{ex\text{-}IF} + (\tau + t_a) = 3\tau + t_a$$

$$T_{ex\text{-}IFM2} = T_{ex\text{-}IF} + 2\ (\tau + t_a) = 4\tau + 2\ t_a$$

Istruzioni aritmetico logiche con *un solo* operando in memoria, e l'altro in registro oppure costante, potrebbero essere del tipo:

COPM11 cost, Rbase, Rindice, Rrisultato

COPM12 Rbase, Rindice, Rsecondo_operando, Rrisultato

rappresentabili su singola parola. Il loro tempo medio di esecuzione è valutato come3:

$$T_{ex\text{-}ALM1} = T_{ex\text{-}ARIT} + (\tau + t_a)$$

Le istruzioni con *due* operandi in memoria sono invece del tipo:

COPM2 Rindirizzo1, Rindirizzo2, Rrisultato

(valgono le stesse considerazioni fatte sopra per le IFM2 circa il formato) ottenendo

$$T_{ex\text{-}ALM2} = T_{ex\text{-}ARIT} + 2\ (\tau + t_a)$$

Avendo introdotto istruzioni più complesse (con maggior tempo medio di elaborazione), ed avendo queste una probabilità non nulla, il tempo medio di elaborazione aumenta, e quindi la performance diminuisce. *Si noti che questa considerazione ha il solo scopo di verificare la comprensione del modo di valutare la performance, non quello di confrontare calcolatori con macchina assembler diversa.*

Fin qui, la Domanda 2.b contiene l'applicazione di concetti e tecniche di base relativi alla macchina assembler ed alla macchina firmware di un calcolatore.

Invece, l'analisi della convenienza o meno delle nuove istruzioni con operandi in memoria, agli effetti della compilazione efficiente di programmi, risulta più delicata. Dal punto di vista didattico e formativo, però, questa parte della domanda è importante, in quanto permette di approfondire la problematica "Risc *vs* Cisc" in maniera non superficiale.

Nonostante quanto si può credere a prima vista, non è detto che la compilazione risulterebbe facilitata o più efficiente usando istruzioni con operandi in memoria.

In ogni caso, la presenza di istruzioni che operano solo su registri, così come di istruzioni LOAD e STORE, è comunque fondamentale per non rendere eccessivamente complicata ed inefficiente la compilazione.

Nel nostro caso, la compilazione del loop della procedura diventerebbe:

```
LOOP:   IFM1>=0   RA, Ri, THEN
        SUBM11    0, RA, Ri, Ra
        GOTO   CONT1
THEN:   LOAD   RA, Ri, Ra
CONT1:  MOD    Ra, RN, Rj
        ADD    RC, Ri, RindC
        ADD    RB, Rj, RindB
        IFM2>=   RindC, RindB, CONT2
        LOAD   RB, Rj, Rb
        STORE   RC, Ri, Rb
CONT2:  INCR   Ri
        IF<   Ri, RN, LOOP
```

Per la parte di calcolo dell'indice *j*, occorre fare accesso in memoria ad A[i] sia nella IFM1 che nella SUB, e comunque trasformare la struttura *if-then* in un *if-then-else*. Le istruzioni LOAD e MOD registro-registro sono comunque presenti.

Per la parte di calcolo del massimo, occorre prima calcolare gli indirizzi con due ADD apposite per poter usare l'istruzione IFM2, ed occorre effettuare un accesso in memoria in più per effettuare la LOAD nel caso che il massimo sia B[j]. La STORE è comunque presente.

Considerando il ramo del loop con tutti i predicati non verificati:

ifm1; subm11; goto; mod; add, add; ifm2; load; store; incr; if

per ogni iterazione vengono effettuati complessivamente (tra istruzioni e dati) 16 accessi in memoria, contro i 14 accessi in memoria del programma compilato con istruzioni "Risc puro" nella Domanda 1.a. Di conseguenza, il tempo di completamento aumenta.

Anche se è possibile cercare ottimizzazioni ad hoc per la compilazione utilizzando istruzioni con operandi in memoria (come nelle macchine Cisc esistenti), l'esempio mostra come il vantaggio di avere istruzioni "potenti" può essere più apparente che reale, e comunque non è affatto ovvio.

Poiché la macchina assembler contiene *tanto* le istruzioni "Risc puro" *quanto* le istruzioni con operandi in memoria, *per questo programma* risulterebbe più efficiente usare le sole istruzioni "Risc puro" come nella Domanda 1.a. In ogni caso, l'analisi di quali istruzioni usare va effettuata *programma per programma,* per cui la complessità del compilatore aumenta, a causa del maggior numero di soluzioni alternative possibili per ogni struttura di programma.

Domanda 3

1. *Un programma assembler A viene interpretato da un proprio microprogramma, diverso da quello che interpreta un programma assembler B.*

Falso. *Per definizione*, l'interprete di un linguaggio L è sempre lo stesso per qualunque programma scritto in L. (Diverso sarebbe il discorso parlando di compilazione, in quanto la compilazione dipende dalle caratteristiche dei singoli programmi scritti nello stesso linguaggio).

Nel caso che L sia il linguaggio della macchina assembler, il concetto è addirittura "tangibile": l'interprete microprogrammato descrive le caratteristiche ed il funzionamento del processore, che ovviamente è sempre lo stesso qualunque sia il programma eseguito.

2. *In un elaboratore general purpose con capacità massima di memoria principale di 1G parole, gli indirizzi generati in istruzioni di Load e Store sono espressi come numeri naturali di 30 bit.*

Falso in generale. Il processore genera indirizzi *logici*, che in generale non coincidono con gli indirizzi fisici.

Nel caso in oggetto, gli indirizzi fisici sono di 30 bit, mentre niente si può dire circa gli indirizzi logici se non sono note le caratteristiche della macchina assembler.

3. *Lo spazio di indirizzamento di un programma è l'insieme di tutti i possibili indirizzi logici che possono essere generati dal programma in esecuzione.*

Vero. Per poter essere eseguito, un programma deve essere in grado di generare tutti gli indirizzi degli oggetti (istruzioni o dati) che, con probabilità non nulla, possono essere da esso riferiti.

Si noti che, se una struttura dati viene allocata nella memoria virtuale, ciò significa che il compilatore assume che ogni sua parte possa essere utilizzata "con probabilità non nulla".

La definizione data include il caso degli oggetti "collegati" a tempo di compilazione (librerie, funzionalità di sistema operativo), e quello degli oggetti creati dinamicamente, per i quali la memoria virtuale viene (almeno in parte) allocata a tempo di esecuzione.

4. *La MMU di un calcolatore implementa una funzionalità di sistema operativo.*

Falso. La funzionalità di traduzione degli indirizzi, affidata (tra le altre) alla MMU, caratterizza il livello assembler-firmware dell'architettura, in quanto permette di riferire informazioni in memoria principale mediante indirizzi logici, quindi indipendenti dall'allocazione della memoria principale stessa.

Ciò non toglie che se (come di regola) il sistema operativo fa uso di allocazione dinamica della memoria principale, allora la MMU costituisca un supporto hardware-firmware indispensabile per ragioni di efficienza; ma l'allocazione dinamica della memoria, e quindi la necessità di una traduzione efficiente degli indirizzi, si può avere anche in una macchina senza sistema operativo.

Osservazioni:

a) una spiegazione come "falso, in quanto la MMU è a livello firmware" è sbagliata. In un sistema a livelli, le funzionalità ad un certo livello sono interpretate ad uno o più dei livelli inferiori. Ad esempio, un programma, applicativo o di sistema, con esigenze particolari di prestazioni può essere implementato a firmware in una unità dedicata (esistono diversi esempi nei sistemi commerciali);

b) la MMU opera su informazioni tratte dalla Tabella di Rilocazione del programma in esecuzione. Tale Tabella viene aggiornata, a tempo di esecuzione, da una funzionalità del sistema operativo e/o del supporto a tempo di esecuzione del linguaggio delle applicazioni (Gestore della Memoria Principale); l'aggiornamento della tabella, però, non è la funzionalità delegata alla MMU;

c) la MMU non ha (non avrebbe senso che avesse) il compito di trasferire dati tra memoria principale e memoria secondaria.

5. *Sostituendo la memoria principale e/o la MMU è necessario modificare i microprogrammi del processore.*

Falso, a condizione che la "nuova" MMU utilizzi un protocollo di comunicazione compatibile con quello adottato nello scrivere il microprogramma del processore.

Certamente nessun impatto hanno le modifiche alla memoria principale, la cui presenza risulta completamente nascosta al processore grazie alla MMU.

Inoltre, nessuna modifica va apportata al processore nel caso cambino le latenze di trasmissione dei collegamenti ed il ciclo di clock della memoria, in quanto *i protocolli di comunicazione sono indipendenti dal tempo*.

Osservazione:

a) quanto sopra permette di utilizzare lo stesso processore con gerarchie di memoria differenti (ad esempio, con o senza cache, vari tipi di cache, ecc.), così come di aggiornare la memoria di un calcolatore senza sostituire il processore.

6. *Un certo algoritmo che opera su array è implementato in due modi diversi: a) a livello assembler per un calcolatore general-purpose, b) a livello firmware in una unità di elaborazione U, avente lo stesso ciclo di clock del processore del calcolatore di cui al punto a) e collegata ad una memoria esterna avente le stesse prestazioni della memoria principale del calcolatore di cui al punto a). L'affermazione è : il tempo di completamento nel caso a) è maggiore di quello del caso b) e la differenza è proporzionale al tempo di accesso in memoria moltiplicato per il numero delle istruzioni eseguite dal programma a).*

Vero. Nel caso *b)* è come avere un processore capace di interpretare, in una sola istruzione, tutta la funzionalità del programma (una sola operazione esterna). Di conseguenza, nel caso *b)* si risparmiano tutti gli accessi in memoria per le *istruzioni*, che invece sono effettuati nel caso *a),* mentre rimangono gli stessi quelli per i dati in quanto, trattandosi di strutture complesse (array), essi sono comunque allocati nella memoria esterna.

Differenze nel numero di cicli di clock non facenti parte di sequenze di accesso in memoria (una costante additiva non nulla può essere presente nell'espressione della differenza dei due tempi di elaborazione) hanno peso in seconda approssimazione.

Osservazioni:

a) un caso in cui viene adottata la soluzione *b)* è nei sistemi o sottosistemi *special-purpose*. Si veda anche l'osservazione a) del punto 4;

b) in sistemi *general-purpose*, la soluzione *b)* può essere attuata disponendo di una tecnologia, che permetta di modificare i microprogrammi del processore (lasciandone inalterata la struttura di PC, PO): un programma verrebbe *compilato in un microprogramma* (eliminando il livello assembler). Tale tecnologia (processori "microprogrammabili") ha avuto una certa popolarità quando il processore veniva implementato su più chip (per permettere di modificare il contenuto di una memoria che implementa le due funzioni ω_{PC}, σ_{PC}). La tecnologia dei processori paralleli (pipeline, superscalari), specie se con assembler Risc, ha di fatto permesso di ottenere le stesse prestazioni conservando il livello assembler, pur di delegare al compilatore un ben più ampio insieme di ottimizzazioni.