Architettura degli Elaboratori - Correzione

Seconda Prova di Verifica Intermedia, a.a. 2005-06, 19 dicembre 2005

Domanda 1

Un elaboratore general-purpose OPS ha il set di istruzioni Risc del capitolo V arricchito dall'istruzione *INCR_MEM Rbase, Rindice*, che ha come effetto di incrementare di uno il contenuto di una locazione di memoria. La cardinalità del set di istruzioni rimane minore o uguale a 256.

La CPU di OPS, con ciclo di clock τ, ha una cache primaria C1 operante su domanda, metodo associativo su insiemi, capacità di 64K parole, ampiezza del blocco di 8 parole, e scritture gestite con il metodo Write-Through. OPS ha una cache secondaria C2 di capacità 1M parole, interallacciata con 4 moduli collegati direttamente alla CPU, ciclo di clock uguale a 4τ, e latenza di trasmissione dei collegamenti inter-chip di 5τ.

a) Scrivere l'interprete di *INCR_MEM*, inclusa la fase di chiamata e decodifica, e valutarne il tempo medio di elaborazione in funzione di τ e del tempo di accesso in memoria t_a.

b) Si consideri un programma applicativo che opera su due array *A, B,* ognuno di *N* interi, con *N* dell'ordine delle migliaia. Per ogni *i = 0, ..., N-1, A[i]* è incrementato di uno se *F(A[i]) > F(B[i])*, con *F* funzione intera disponibile come procedura.

La procedura *F* consta di 10 istruzioni, opera solo su dati in registri generali, ed ha tempo medio di completamento uguale a 50τ. I parametri di ingresso e di uscita sono passati per valore attraverso registri generali.

Si indichi con *p* la probabilità che *F(A[i]) > F(B[i])* per qualsiasi *i.*

Valutare, in funzione di τ, *p* e *N*, il *tempo medio di completamento* del programma *su OPS*, supponendo trascurabile la probabilità di fault della gerarchia di memoria M-C2, con M memoria principale.

Dire e spiegare quanto è ampio *l'insieme di lavoro (working set)* del programma.

c) Dire se la seguente affermazione è vera o falsa, spiegando la risposta: "se C1 rileva un fault di blocco, il trattamento di questo evento può generare fault anche in M".

Domanda 2

Il programma della Domanda 1 è completato in modo che gli array *A, B* siano letti da un certo dispositivo D e che l'array *A* modificato sia scritto sullo stesso dispositivo.

A livello del linguaggio delle applicazioni, D è visto attraverso i comandi *get(n, x)* e *put (n, x),* dove *x* è il nome di una variabile composta da *n* byte; *get* è il comando per ottenere *x*, e *put* il comando per inviare *x*.

Il linguaggio concorrente L_c con cui è scritto il sistema operativo di OPS è quello a scambio di messaggi delle "Note sul livello dei processi".

a) Mostrare e spiegare la *compilazione in* L_c del programma *in un processo* APPL, supponendo che all'unità I/O_D sia associato un processo Driver_D. Per semplicità, non si considerino eccezioni generate dalle operazioni *get* e *put*.

b) Spiegare se la versione *assembler* di APPL va modificata o meno, ed eventualmente come, nel caso che sia eliminato il processo Driver_D, e se, in tale versione, è visibile ad APPL quale modello, DMA o Memory Mapped I/O, è adottato per i trasferimenti di I/O.

c) Spiegare in seguito a quali eventi il processo APPL può *transire in stato di attesa*, ed in seguito a quali eventi può essere *risvegliato*, considerando esplicitamente anche il caso in cui non esiste il processo Driver_D.

Domanda 1

a) L'interprete richiesto è espresso dal seguente microprogramma; la fase di chiamata istruzione e decodifica fa uso della tecnica del *salto forzato* mediante la funzione identità per la decodifica del codice operativo; per richiedere la lettura dell'istruzione, viene segnalata l'operazione *'execute'* affinché la MMU distingua il tipo di oggetto (in questo caso, "codice") per ragioni di protezione (l'utilizzo dell'operazione *'read'* è considerato comunque corretto). La fase di esecuzione consiste nelle lettura, incremento e scrittura allo stesso indirizzo:

{chiamata istruzione e decodifica}

ch0. IC → IND, 'execute' → OP, set RDYOUT, ch1

ch1. (RDYIN, or(ESITO) = 0 -) nop, ch1;

(= 1 1) reset RDYIN, ESITO → ESITO1, tratt_ecc ;

(= 1 0) reset RDYIN, DATAIN → IR, *DATAIN.COP → RC*

Formato istruzione su singola parola:

COP (8 bit)
Rbase (6)
Rindice (6)
Per usi speciali (≥ 3)
Non specificati (≤ 9)

{esecuzione INCR_MEM}

im0. RG [IR.Rbase] + RG [IR.Rindice] → IND, 'read' → OP, set RDYOUT, im1

im1. (RDYIN, or(ESITO)= 0 -) nop, im1;

(= 1 1) reset RDYIN, ESITO → ESITO1, tratt_ecc ;

(= 1 0) reset RDYIN, DATAIN + 1 → DATAOUT, 'write' → OP, set RDYOUT, im2 // IND è inalterato //

im2. (RDYIN, or(ESITO), INT = 0 - -) nop, im2;

(= 1 1 -) reset RDYIN, ESITO → ESITO1, tratt_ecc ;

(= 1 0 0) reset RDYIN, IC + 1 → IC, ch0;

(= 1 0 1) reset RDYIN, IC + 1 → IC, tratt_int

Il tempo medio di elaborazione è dato da:

$$T_{INCR\_MEM} = T_{ch} + T_{ex\_INCR\_MEM} = (2\tau + t_a) + (3\tau + 2t_a) = 5\tau + 3t_a$$

Osserviamo che, rispetto alla stessa funzionalità espressa a livello assembler mediante l'assembler Risc del Cap. V, viene risparmiato un tempo dell'ordine di $2t_a$ (per l'esattezza: $6\tau + 2t_a$).

b) Il programma

```
int A[N], B[N];
for (i = 0; i < N; i++)
        if F(A[i]) > F(B[i])
                A[i] = A[i] + 1;
```

è compilato come mostrato accanto, utilizzando la regola di compilazione del *for* con almeno una iterazione.

Il compilatore provvede a inizializzare: RG[RA] e RG[RB] rispettivamente all'indirizzo logico base di *A* e *B*; RG[Ri] a zero; RG[RN] alla costante *N*; RG[RF] all'indirizzo della procedura *F*. Il parametro formale d'ingresso alla procedura è il contenuto del registro di indirizzo RFin, quello di uscita RFout. I registri di indirizzo Ra, Ra1, Rret sono allocati come temporanei.

```
LOOP:   LOAD   RA, Ri, Ra
        MOVE   Ra, RFin
        CALL   RF, Rret
        MOVE   RFout, Ra1
        LOAD   Rb, Ri, RFin
        CALL   RF, Rret
        IF<=   Ra1, RFout, CONT
        INCR   Ra
        STORE  RA, Ri, Ra
CONT:   INCR   Ri
        IF<    Ri, RN, LOOP
        END
```

Essendo la cache C1 operante *su domanda*, il trasferimento di un blocco da C2 viene effettuato ad ogni fault sequenzialmente all'elaborazione. Il tempo di completamento si valuta come:

$$T_c = T_{c-id} + T_{fault} = T_{c-id} + N_{fault} * T_{trasf}$$

dove

- $T_{c\text{-}id}$ è il tempo di completamento ideale, in assenza di fault. Esso è quindi valutando assumendo che il tempo di accesso alla memoria sia uguale al tempo di accesso a C1; nel nostro caso, essendo usato il metodo completamente associativo, $t_c = 3\tau$;
- N_{fault} è il numero di fault di C1 che si verificano durante l'esecuzione del programma. Le istruzioni (in numero di 22, inclusa la procedura *F*) occupano in tutto tre blocchi di cache e provocano solo tre fault, dopo di che, grazie alla proprietà del riuso, esse sono sempre presenti in cache; si può dunque trascurare la probabilità di fault delle istruzioni. Per i dati, si verifica un fault di *A* e di *B* ogni σ iterazioni, quindi

$$N_{fault} = \frac{2N}{\sigma}$$

- T_{trasf} è il tempo per trasferire un blocco da C2 a C1. Essendo C2 interallacciata con $m = 4$ moduli, occorrono due letture da C2 per un blocco di $\sigma = 8$ parole, delle quali la seconda lettura si sovrappone alla scrittura in C1 delle 4 parole ottenute con la prima lettura:

$$T_{trasf} = \frac{\sigma}{m} t_{C2} + m\tau = 2t_{C2} + 4\tau = 2(\tau_{C2} + 2T_{tr}) + 4\tau = 32\tau$$

Quindi:

$$T_{fault} = N_{fault} * T_{trasf} = 8N\tau$$

Questo risultato è valido assicurando che, istante per istante, C1 contenga *l'insieme di lavoro* del programma, eccetto che quando si verificano fault con la probabilità vista in precedenza. L'insieme di lavoro consta di 5 blocchi: tre per le istruzioni, uno per *A* e uno per *B*. Una volta caricati, i tre blocchi delle istruzioni non vengono più scaricati. Quindi, dato il rapporto tra capacità di C1 e ampiezza del blocco, l'insieme di lavoro risiede ampiamente in C1, eccetto quando si verificano fault di *A* e di *B*. Data la flessibilità del metodo associativo su insiemi, blocchi corrispondenti di *A* e *B* risiedono contemporaneamente in C1: è sufficiente che il nuovo blocco di *B* sia allocato in un qualunque blocco inutilizzato oppure che non sostituisca il nuovo blocco di *A*.

La scrittura nella cache secondaria (metodo *Write Through*) non ha influenza sul tempo di completamento, in quanto sarebbe mascherata completamente dall'esecuzione di una iterazione del programma, persino nel caso di organizzazione sequenziale di C2; in realtà, essendo C2 interallacciata, accessi consecutivi avvengono a moduli di consecutivi, quindi, se necessario, sarebbero comunque effettuati in parallelo.

Utilizzando i tempi medi di elaborazione delle istruzioni del linguaggio assembler, il tempo di completamento ideale è dato da:

$$T_{c\text{-}id} = N\,[\,(9 + 2p)\,T_{ch} + (2 + p)\,T_{ex\_LD/ST} + 2\,T_{ex\_CALL} + 2\,T_{ex\_MOVE} + 2\,T_{ex\_IF} + (1 + p)\,T_{ex\_INCR} + 2\,T_{c\_F}\,] =$$

$$= N\,[\,(9 + 2p)\,(2\tau + 3\tau) + (2 + p)\,(2\tau + 3\tau) + 2\,(\tau) + 2\,(\tau) + 2\,(2\tau) + (1 + p)\,(\tau) + 100\tau\,] = (164 + 11\,p)\,N\tau$$

Quindi:

$$T_c = T_{c-id} + T_{fault} = (172 + 16p)N\tau$$

c) L'affermazione è falsa, alla luce del significato e del funzionamento di ogni gerarchia di memoria:), *le richieste di accesso contengono l'indirizzo del supporto del livello più alto e sono dirette al supporto del livello più basso*. Nella gerarchia "memoria principale – cache" (indipendentemente dall'esistenza di livelli intermedi di cache), le richieste di accesso contengono l'indirizzo di M e sono dirette a C1*:* la richiesta a C1 è effettuata dalla MMU che ha provveduto a tradurre l'indirizzo logico del programma in indirizzo fisico di M. Questa traduzione è avvenuta certamente *con successo*: infatti, se l'informazione cercata non risiedesse in M, MMU avrebbe rilevato un fault della gerarchia "memoria virtuale – memoria principale" e non avrebbe effettuato alcuna richiesta a C1 (avrebbe segnalato direttamente l'eccezione al processore).

(D'altra parte, non è possibile che avvenga una diversa allocazione di M senza che, in maniera indivisibile, sia aggiornata la Tabella di Rilocazione del processo in esecuzione, e quindi senza che sia noto a MMU.)

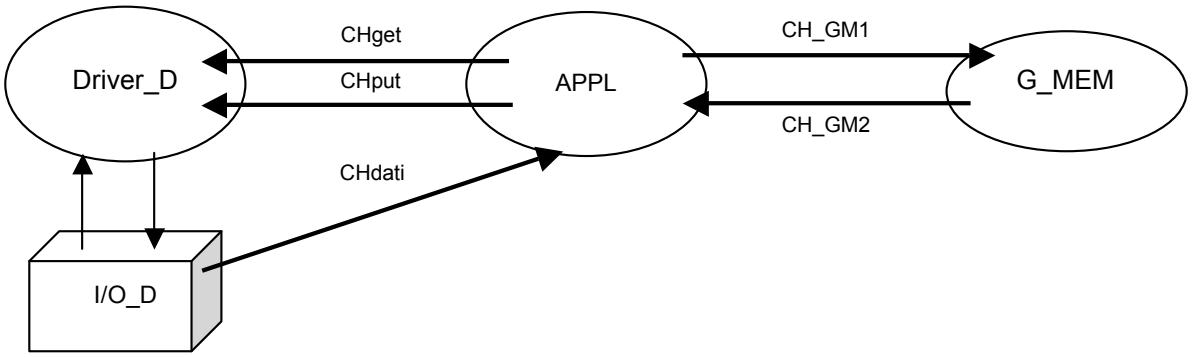
Domanda 2

a) Il programma applicativo sorgente è il seguente:

```
int A[N], B[N];
    {   get (N, A);
    {   get (N, B);
        for (i = 0; i < N; i++)
                if F(A[i]) > F(B[i])
                        A[i] = A[i] + 1;
        put (N, A)
    }
```

Il processo Driver_D dispone di due canali (asimmetrici) d'ingresso, di identificatori *CHget* e *CHput*, corrispondenti alle due operazioni da esso implementate per virtualizzare D: attraverso *CHget* riceve la dimensione del blocco di dati e l'identificatore del canale *CHdati* su cui l'applicazione si aspetta di ricevere il blocco di dati (il cui valore verrà assegnato ad *A* oppure a *B*); attraverso *CHput* riceve la dimensione ed il valore del blocco di dati da inviare a D.

Lo schema a processi comunicanti della computazione L_c è mostrato nella figura seguente:



Per minimizzare il tempo di comunicazione dei valori degli array *A* e *B*, il canale *CHdati* ha come mittente direttamente il *processo esterno* I/O_D. A questo scopo, il valore dell'identificatore *CHdati*, che Driver_D ha ricevuto in una variabile di tipo *channelname*, viene inoltrato da Driver_D ad I/O_D con il messaggio di richiesta di lettura del blocco. Poiché quando I/O_D eseguità una *send* su *CHdati* troverà APPL in attesa, se (come di regola) l'implementazione delle *send-receive* fa uso di ottimizzazioni, la copia del valore di *A* o di *B* avverrà direttamente in tali variabili, minimizzando complessivamente il numero di copie.

Oltre ai canali di comunicazione con Driver_D e I/O_D, APPL utilizza i canali per il *trattamento dell'eccezione di fault di pagina* (*CH_GM1, CH_GM2*), mediante i quali comunica al processo gestore della memoria principale i parametri relativi ad una situazione di fault di pagina, ed attende la segnalazione che la pagina richiesta sia stata allocata e trasferita in M e che la Tabella di Rilocazione sia stata aggiornata.

Tutti i canali di ingresso di APPL sono asincroni di una posizione: ciò è sufficiente ad assicurare che i mittenti delle comunicazioni su tali canali non si sospenderanno mai nell'esecuzione delle *send* relative, trattandosi di colloqui a domanda e risposta. Pur non essendo visibile in APPL, anche i canali di ingresso degli altri processi conviene che siano asincroni.

La compilazione del processo applicativo nel linguaggio L_c è:

APPL:: **channel in** CHdati (1), CH_GM2 (1); **channel out** CHget, CHput, CH_GM1; <altre dichiarazioni>;

```
{   send ( CHget, (Chdati, 4*N) );
    receive ( CHdati, A );
    send ( CHget, (Chdati, 4*N) );
    receive ( CHdati, B );
    for (i = 0; i < N; i++)
        if F(A[i]) > F(B[i])
            A[i] = A[i] + 1;
    send ( CHput, (4*N, A) )
}
```

Aggancio alla gestione di D per ottenere le N parole di A e le N parole di B

Algoritmo dell'applicazione

Aggancio alla gestione di D per inviare le N parole di ^

```
{   Handler per trattamento interruzioni;
    Handler per trattamento eccezioni (CH_GM1, CH_GM2)
}
```

La compilazione *in assembler* produce, oltre al codice visto nella Domanda 1, il codice del supporto a tempo di esecuzione di L_c: supporto delle primitive *send* e *receive*, incluso lo scheduling a basso livello, trattamento di interruzioni ed eccezioni. Il compilazione inizializza tutte le strutture dati del supporto di L_c.

b) Eliminando il processo Driver_D, *non va apportata alcuna modifica* alla versione assembler di APPL, in quanto valgono *entrambe* le seguenti condizioni:

1) l'unità di I/O è vista come un processo;

2) tutti i dettagli dell'implementazione dei trasferimenti di dati e della sincronizzazione tra processi, *inclusi i trasferimenti di I/O*, sono completamente trattati nel supporto delle primitive *send* e *receive*.

Ora, i canali *CHget* e *CHput* saranno in ingresso al processo esterno I/O_D, senza che ciò sia minimamente visibile ad APPL: le primitive *send e receive* lavorano su descrittori di canale, senza alcuna conoscenza del nome del processo partner né tanto meno della sua implementazione. Quando si tratti di eseguire operazioni di "sveglia partner", queste riferiscono indirizzi di PCB presenti nel descrittore di canale.

Il modello dei trasferimenti di I/O (DMA o Memory Mapped I/O) è *completamente invisibile* nel codice di APPL, in quanto valgono *entrambe* le seguenti condizioni:

1) tutti i dettagli dell'implementazione dei trasferimenti di dati e della sincronizzazione tra processi, *inclusi i trasferimenti di I/O*, sono completamente trattati nel supporto delle primitive *send* e *receive*;

2) le primitive *send* e *receive* utilizzano indirizzi logici, senza fare alcuna assunzione sul fatto che gli oggetti riferiti (descrittori di canali, messaggi, variabili targa, ecc.) siano allocati fisicamente in memoria principale o nelle memorie di I/O.

c) Il processo APPL non può mai sospendersi sulla primitiva *send(CHget, ...),* a condizione che *CHget* sia asincrono (è sufficiente il grado di asincronia uguale a uno), visto che il colloquio per implementare il comando *get* è a domanda e risposta. Lo stesso vale per la *send(CH_GM1, ...).*

APPL può sospendersi (si sospende) sulle primitive *receive(CHdati, ...)* e *receive(CH_GM2, ...)*, proprio in quanto il colloquio è a domanda e risposta.

APPL può inoltre sospendersi sulla primitiva *send(CHput, ...),* se l'esecuzione di tale *send* è ripetuta un numero di volte maggiore del grado di asincronia di *CHput* senza che il partner abbia ancora eseguito una *receive* corrispondente.

APPL verrà risvegliato dai partner delle comunicazioni, su cui si è sospeso, quando questi eseguiranno le primitive corrispondenti. Nel caso che il partner sia il processo esterno I/O_D, la sveglia di APPL viene effettuata *in seguito all'interruzione* che I/O_D invia quando, *e se*, rileva (attraverso informazioni nel

descrittore di canale *CHdati*: campo *wait*) che APPL si trova in attesa. Il processo attualmente in esecuzione sul processore, una volta completata la fase firmware del trattamento interruzioni, esegue l'Handler che consiste proprio nella procedura di sveglia processo.