Architettura degli Elaboratori, 2008-09

Esercitazione 3

Ogni esercitazione ha lo scopo di servire da guida per la preparazione su una specifica parte del corso. È fortemente consigliato che lo studente lavori indipendentemente all'esercitazione durante lo svolgimento di tale parte a lezione e prima che sia disponibile la soluzione, approfondendo criticamente i vari aspetti e accompagnando la soluzione con adeguate spiegazioni rivolte alla comprensione ed alla esposizione dei concetti del corso.

Soluzione: venerdì 5 dicembre

Domanda 1

Si consideri un programma che opera su tre array di interi *A[N]*, *B[N]*, *C[N]*, con N = 16K, nel seguente modo:

$$\forall\ i = 0\ ..\ N–1\text{: } C[i] = \text{numero di volte che } A[i] \text{ compare in } B$$

Compilare il programma in assembler D-RISC, strutturandolo in modo che il numero di volte che un valore intero compare in un array di N interi sia implementato come una procedura. Il passaggio dei parametri alla procedura deve essere effettuato via memoria.

Spiegare come sono stati allocati, ed eventualmente inizializzati, i registri generali, e come sono state applicate le regole di compilazione.

Spiegare quali modi di indirizzamento sono stati utilizzati nella compilazione, distinguendo tra modi primitivi in D-RISC e modi non primitivi, nel secondo caso spiegando come sono stati implementati.

Descrivere la memoria virtuale e lo spazio di indirizzamento del processo corrispondente al programma, spiegando quali locazioni della memoria virtuale sono inizializzate a tempo di compilazione. Si suppone che la memoria virtuale sia organizzata in pagine ognuna di ampiezza 1K parole; qualunque oggetto (codice o dato) deve essere allocato in un numero intero di pagine, eventualmente sprecando parti di pagine. Si suppone anche che gli oggetti da "collegare" al processo (PCB e altri meccanismi per supportare il concetto di processo e sua cooperazione con altri processi) occupino complessivamente 10K parole.

Domanda 2

Valutare il tempo di completamento del programma della Domanda 1 per una architettura con frequenza del clock della CPU di 4 GHz, memoria principale avente ciclo di clock uguale a 50 volte il ciclo di clock τ della CPU, collegamenti inter-chip con latenza di trasmissione uguale a 10 τ.

Utilizzare questo programma per valutare la performance dell'architettura.

Individuare possibili caratteristiche di un set di istruzioni, più complesso di D-RISC, che permetta di ottenere un tempo di completamento inferiore, e spiegare come queste caratteristiche sono implementate.

Domanda 3

Valutare il tempo di completamento del programma della Domanda 1 per una architettura la cui CPU, rispetto alla Domanda 2, possiede anche una cache primaria operante su domanda, completamente associativa, di capacità di 64K parole, blocchi di 8 parole, scritture con il metodo Write-Through. Non è presente cache secondaria. La memoria principale, oltre ad avere le caratteristiche della Domanda 2, è interallacciata con 4 moduli.

Nel dare la valutazione, spiegare come si caratterizza il programma dal punto di vista della località e del riuso, e determinarne l'insieme di lavoro. Spiegare *se* e *come* per garantire la presenza di tale insieme di lavoro è necessario intervenire a tempo di compilazione, oppure a tempo di esecuzione.

Soluzione

Domanda 1

Per poter discutere le problematiche di compilazione, definiamo un programma C che implementi le specifiche dell'algoritmo della Domanda 1. Il programma è composto, oltre che dal programma principale (*main* in C), anche da una procedura che, come da specifiche dell'esercizio, implementa il calcolo del numero di volte che un valore intero (primo parametro) è presente in un array di N interi (secondo parametro). La procedura restituisce tale numero. Implementiamo la procedura in un file apposito *conta_volte.c* che verrà poi appositamente "collegato" al file in cui implementiamo la procedura principale durante la fase di compilazione.

```
#define K 1024
#define N 16 * K

int conta_volte(int val, int arr[N]) {
     int i, volte = 0;

     for(i = 0; i < N; i++)
         if(val == arr[i]) volte++;

    return volte;
}
```

Nota: per verificarne l'aderenza con lo standard C, i programmi sono stati compilati in ambiente Linux con compilatore gcc versione 4.1.3 con opzioni –Wall –pedantic –ansi. Alcune convenzioni proprie del linguaggio C (e.g. il main restituisce un intero, vedi sotto) non avranno corrispondenza nella compilazione in D-RISC.

La costante *N* è una *macro* del programma con valore 16 · 1024 (16K). La procedura *conta_volte* scorre il secondo parametro formale (*arr*) alla ricerca del valore contenuto nel primo parametro formale (*val*). Questa procedura è da considerarsi alla stessa stregua di una libreria (composta da un'unica funzione): il programmatore ha a disposizione una procedura già implementata che conta quante volte il primo parametro è contenuto nel secondo parametro, la cui lunghezza è fissata (*N*). Supponiamo che il file *conta_volte.h* contenga la dichiarazione del prototipo della procedura *conta_volte*. Vediamo adesso il programma principale che fa utilizzo di tale procedura e che implementiamo nel file *mymain.c*.

```
#include "conta_volte.h"

#define K 1024
#define N 16 * K

int main(int argc , char ** argv) {
    int A[N], B[N], C[N];
    int i;

    for(i = 0; i < N; i++)
        C[i] = conta_volte(A[i], B);

   return 0;
}
```

Nel programma principale ridichiariamo la costante *N* e, scorrendo gli array A e C, invochiamo la procedura *conta_volte*. Per compilare questo programma possiamo passare all'eseguibile del compilatore i due file (*conta_volte.c* e *mymain.c*).

Nota: in questa soluzione diamo la procedura di ragionamento completa per la compilazione del programma: queste spiegazioni sono volutamente sovrabbondanti rispetto a quelle richieste normalmente in fase di esame scritto, ma sono date al fine di semplificare e rendere più efficace la metodologia di esposizione degli studenti. La procedura di compilazione del programma è relativa alla fase di generazione del codice che corrisponde tipicamente al "back-end" di compilatori noti: assumiamo di non dover

controllare la correttezza sintattica del programma. Tale parte, classicamente associata alla fase di front-end del compilatore (parser), sarà oggetto di corsi avanzati della Laurea in Informatica. È inoltre importante far notare allo studente che studia queste soluzioni che la terminologia relativa ai registri e alle locazioni di memoria e alle loro interrelazioni (es.: il registro generale di indirizzo R_A contiene l'indirizzo base di memoria virtuale dell'array A) è un punto critico di questa parte del corso e che è soggetto a valutazione. La distinzione fra indirizzi e loro contenuti, anche nel linguaggio parlato e scritto, è un bagaglio indispensabile per la comprensione di tutti i linguaggi di programmazione, sia che essi siano oggetto di corsi all'interno della laurea, sia che non lo siano.

Come abbiamo visto, il compilatore conosce il valore di *N*, che corrisponde a 16K [1]. Gli array A, B e C nella procedura principale (*main*) contengono 16384 elementi interi (vedi dichiarazione: int A[N]). La variabile *i* è unicamente utilizzata, in entrambe le procedure, come indice di ciclo. Quindi, sebbene ogni variabile debba trovare spazio all'interno della Memoria Virtuale (MV), in questo caso possiamo introdurre un'ottimizzazione: implementiamo *i* in un registro (in seguito R_i programma main e R_j nella procedura) senza far uso di memoria virtuale. La variabile *i* è inizializzata a 0 in entrambi i cicli: poiché 0 < 16K, il ciclo *for* verrà eseguito almeno una volta. È quindi possibile scegliere un'implementazione (nel linguaggio che caratterizza l') ASSEMBLER secondo lo schema *do-while*. Questa soluzione è preferibile perché minimizza il numero di salti all'interno del codice: si vedrà in corsi successivi a questo come questo rappresenti un'ottimizzazione nel caso di processori *pipeline* e *superscalari*.

Iniziamo implementando la procedura *conta_volte*. L'implementazione fa uso dei seguenti registri: il registro R_j è utilizzato come indice del ciclo. Il registro R_{macro} contiene il valore di *N*. R_{volte} è utilizzato come accumulatore del numero di volte che *val* compare in *arr* ed è il valore risultato della procedura: verrà copiato in memoria virtuale durante il passaggio dei parametri di uscita alla conclusione della procedura. R_{form1} e R_{form2} contengono gli indirizzi (logici) delle locazioni di memoria in cui il chiamante della procedura copia il valore dei parametri attuali. *Questo deriva dalla richiesta del testo di implementare il passaggio dei parametri via memoria*[2]. Entrambi i registri sono inizializzati in fase di compilazione: il compilatore genera la memoria virtuale del processo e quindi decide quali sono gli indirizzi dei due parametri della procedura *conta_volte*. R_{val} contiene, dopo il passaggio dei parametri, il valore del primo parametro. R_{arr} contiene, sempre dopo il passaggio dei parametri, il valore del secondo parametro. R_{temp} contiene, ad ogni iterazione del ciclo, il valore *arr*[*j*]. R_{ret} contiene l'indirizzo dell'istruzione di ritorno dalla procedura ed è condiviso fra il programma principale e la procedura. R_{form3} contiene l'indirizzo della locazione di memoria che, alla conclusione della procedura, servirà per il passaggio del parametro di ritorno della procedura. Per i motivi spiegati sopra, il ciclo della procedura è implementato secondo uno schema *do-while*.

```
        CLEAR Rj              // 0: 0 → Rj
        CLEAR Rvolte          // 1: 0 → Rvolte
        LOAD Rform1, 0, Rval  // 2: trasferimento del valore del primo parametro da mem. al reg. Rval
        LOAD Rform2, 0, Rarr  // 3: trasferimento del valore del secondo parametro da mem. al reg. Rarr
LOOP:   LOAD Rarr, Rj Rtemp   // 4: trasferimento di arr[j] da mem. al reg. Rtemp
        IF ≠ Rval, Rtemp, CONT // 5: controllo se i due valori caricati sono uguali
        INCR Rvolte           // 6: se i due valori sono uguali incremento il contatore
CONT:   INCR Rj               // 7: incremento il registro di controllo del ciclo
        IF < Rj, Rmacro, LOOP // 8: controllo la terminazione del ciclo
        STORE Rform3, 0, Rvolte // 9: restituisco il valore calcolato via memoria
        GOTO Rret             // 10: salto all'istruzione successiva a quella di chiamata della proc.
```

Lettura dei parametri in ingresso da MV.

Scrittura del parametro in uscita alla procedura in MV.

[1] Nel caso del compilatore gcc, un precompilatore precede la fase di controllo di correttezza sintattica. In tale fase di precompilazione tutte le costanti definite come macro sono sostituite. Il generatore del codice vedrà quindi la dichiarazione di A come: `int A[16384]`.

[2] Altro discorso sarebbe stato se il testo dell'esercizio avesse chiesto di implementare il passaggio dei parametri tramite registri generali! Per questo tipo di passaggio si vedano le esercitazioni svolte in aula.

Passiamo quindi a compilare il programma principale. I registri utilizzati sono i seguenti: R_{att1} e R_{att2} contengono gli indirizzi dei parametri passati alla procedura *conta_volte*. Il loro contenuto sarà quindi lo stesso dei registri R_{form1} e R_{form2} utilizzati nell'implementazione della procedura[3]. Come tali registri, anche questi sono inizializzati dal compilatore. R_A, R_B e R_C contengono rispettivamente gli indirizzi base dei tre array A, B e C e sono inizializzati in fase di compilazione. R_i viene utilizzato come indice per il controllo della terminazione del ciclo e per l'indirizzamento degli array. È quindi inizializzato a 0 dal compilatore. R_N contiene il valore della costante *N* ed è inizializzato in fase di compilazione. Il suo valore è lo stesso del registro R_{macro} utilizzato nell'implementazione della procedura. R_a è utilizzato per mantenere il valore A[i] ad ogni ciclo del for nel programma principale. R_{conta_volte} contiene l'indirizzo della procedura *conta_volte* ed è inizializzato in fase di compilazione. R_{ret} è utilizzato per memorizzare l'indirizzo dell'istruzione di ritorno al momento della chiamata di procedura. R_{att3} contiene l'indirizzo di memoria virtuale in cui la procedura copia il risultato. Il valore contenuto in questo registro è lo stesso di quello contenuto in R_{form3} utilizzato dalla procedura. Come tale registro, anche questo è inizializzato in fase di compilazione. R_c è utilizzato per mantenere il valore restituito dalla procedura e copiarlo in C[i], ad ogni ciclo. Il ciclo del programma principale è implementato secondo uno schema *do-while* per le motivazioni viste sopra.

```
LOOP1: LOAD R_A, R_i, R_a     // 0: trasferisco il valore A[i] dalla memoria a R_a
       STORE R_att1, 0, R_a   // 1: trasferisco il secondo parametro (A[i]) via memoria
       STORE R_att2, 0, R_B   // 2: trasferisco l'ind. dell'array B nella loc.di mem. per il secondo parametro della proc.
       CALL R_conta_volte, R_ret // 3: chiamo la procedura conta_volte
       LOAD R_att3, 0, R_c    // 4: traferisco il ris. della procedura da memoria a R_c
       STORE R_C, R_i, R_c    // 5: trasferisco il risultato della procedura dal reg. R_c alla loc. di memoria contentente C[i]
       INCR R_i               // 6: incremento il reg. contatore del ciclo
       IF < R_i, R_N, LOOP1   // 7: controllo la terminazione del ciclo for
       END                    // 8: terminazione del processo
```

Passaggio dei parametri in ingresso e in uscita della procedura

Si noti che i due array A e B non sono "riempiti" all'inizio del programma. Si assuma che questo sia solo la parte "interessante" del programma, dove abbiamo tralasciato l'acquisizione dei dati di ingresso.

Il numero dei registri utilizzati nel programma principale sommati a quelli utilizzati nella procedura è inferiore a 64. Altrimenti avremmo dovuto introdurre delle ottimizzazioni (e.g. R_{macro} e R_N condivisi fra programma principale e procedura) o mantenere in memoria il valore dei registri della procedura.

Infine, analizzando il codice C del programma principale possiamo notare come il secondo parametro della procedura *conta_volte* sia lo stesso per tutta la durata del ciclo (e quindi del programma). Un'ottimizzazione consiste nell'eseguire la STORE del secondo parametro alla linea 2 una sola volta (e non *N*), prima dell'istruzione etichettata con LOOP. In questo caso, si tratta di capire sotto quali condizioni sono possibili tali ottimizzazioni statiche.

Metodi di indirizzamento utilizzati

Nell'implementazione del programma si è fatto uso dei seguenti metodi di indirizzamento:

- *Indirizzamento assoluto dei registri generali* (primitivo in D-RISC): nelle istruzioni si utilizzano indirizzi di registri su 6 bit (64 registri generali). Ad esempio un'istruzione che utilizza questo tipo di indirizzamento è: INCR R_i
- *Indirizzamento base+indice* (primitivo): utilizzato per accedere alle posizioni dei tre array A, B e C. Un esempio è l'istruzione: LOAD R_A, R_i R_a, dove R_A e R_i contengono rispettivamente l'indirizzo logico base e l'offset.
- *Indirizzamento indiretto via registro* (primitivo), ad esempio STORE R_{att1}, 0, R_a e GOTO R_{ret}:

[3] Nel caso in cui l'implementazione utilizzasse un numero di registri superiore a 64, un'ottimizzazione consisterebbe nel far condividere i registri per il passaggio dei parametri in ingresso e in un'uscita fra la procedura e il programma principale.

- *Indirizzamento relativo* (primitivo) per le istruzioni condizionali. Ad esempio nella procedura *conta_volte* nell'istruzione IF ≠ R_{val}, R_{temp}, CONT il valore di CONT è +2, indicando che si deve saltare a due istruzioni sotto (nell'implementazione: IC = IC + 2).
- *Indirizzamento indiretto via memoria* (non primitivo): è utilizzato per il passaggio *per riferimento* del secondo parametro alla procedura *conta_volte*: LOAD R_{form2}, 0, R_{arr} copia il contenuto della locazione di memoria (il cui indirizzo è contenuto in R_{form2}) nel registro R_{arr}. Tale contenuto rappresenta l'indirizzo base dell'array passato come secondo parametro alla procedura che è quindi successivamente utilizzato come base per caricare i vari elementi dell'array nell'istruzione LOAD R_{arr}, R_j R_{temp}.

Organizzazione della memoria virtuale

Vediamo in che modo, a partire dal programma C, il compilatore può generare un file oggetto in cui sia descritto tutta l'implementazione a livello ASSEMBLER del programma. La memoria virtuale contiene i seguenti oggetti agli indirizzi indicati:

- Indirizzi 0..16K-1: array A non inizializzato.
- Indirizzi 16K..32K-1: array B non inizializzato.
- Indirizzi 32K..48K-1: array C non inizializzato.
- Indirizzo 48K: primo parametro in ingresso della procedura *conta_volte*.
- Indirizzo 48K+1: secondo parametro in ingresso della procedura *conta_volte*.
- Indirizzo 48K+2: terzo parametro in uscita della procedura *conta_volte*[4].
- Indirizzi 49K: codice del programma principale.
- Indirizzi 50K: codice della procedura *conta_volte*.
- Indirizzi 51K..51K+127: PCB del processo. Le informazioni inizializzate a tempo di compilazione sono quelle delle immagini di alcuni registri generali come da spiegazioni date per l'implementazione ASSEMBLER.
- Indirizzi da 52K..61K: altri oggetti da collegare al processo (9K): Tabella di Rilocazione del processo (1 pagina), spazio per i PCB di tutti gli altri processi attivabili, codici e dati locali delle primitive di concorrenza, incluse le funzionalità di scheduling a basso livello, handler di interruzioni ed eccezioni.

[4] I tre parametri della procedura *conta_volte* sono allocati nella stessa pagina.

Domanda 2

Il tempo di completamento è espresso da:

$$T_c \sim N \cdot T_{iter}$$

dove T_{iter} è il tempo necessario ad eseguire un'iterazione del ciclo del programma principale. L'iterazione consiste principalmente nell'esecuzione della procedura *conta_volte*, il cui tempo di elaborazione prevale nettamente su quello del passaggio dei parametri. Quindi:

$$T_c \sim N \cdot T_{proc}$$

dove T_{proc} è il tempo necessario ad eseguire la procedura.

La procedura contiene a sua volta un ciclo eseguito N^2 volte, più una fase di pre- e post-elaborazione (linee 0-3 e 9-10 rispettivamente) che implementano l'acquisizione dei parametri, la preparazione dei registri utilizzati e la restituzione dei risultati:

```
        CLEAR Rj                          }
        CLEAR Rvolte                      }
        LOAD Rform1, 0, Rval              }  PRE
        LOAD Rformr2, 0, Rarr             }
LOOP: LOAD Rarr, Rj Rtemp                 }
        IF ≠ Rval, Rtemp, CONT            }  LOOP
        INCR Rvolte                       }
CONT: INCR Ri                             }
        IF < Rj, Rmacro, LOOP             }
        STORE Rform3, 0, Rvolte           }  POST
        GOTO Rret                         }
```

Poiché il numero di istruzioni che implementano le fasi PRE e POST sono in numero trascurabile rispetto a quelle che implementano la fase LOOP ($N^2 \cdot 5$ contro 6 istruzioni) il tempo di completamento è dominato dal ciclo interno della procedura. Quindi:

$$T_c \sim N^2 \cdot T_{loop}$$

Si ha:

$$T_{loop} = 5 \cdot T_{ch} + T_{load} + 2 \cdot T_{if} + 2 \cdot T_{incr}$$

dove T_{ch} è il tempo medio di chiamata e decodifica di un'istruzione. *Si è supposto che la probabilità che A[i] = B[j] sia trascurabile rispetto al caso complementare.*

Il tempo di accesso in memoria principale è dato da:

$$t_a = \tau_M + 2 \cdot (\tau + T_{tr}) = 72\,\tau$$

Si ha che:

$$T_{ex-INCR} = \tau$$

$$T_{ex-LOAD} = 2\,\tau + t_a = 74\,\tau$$

$$T_{ex-IF} = 2\,\tau$$

$$T_{ch} = 2\,\tau + t_a = 74\,\tau$$

Quindi il tempo di completamento è calcolato in funzione di N e τ come:

$$T_c = N^2 \cdot T_{loop} = N^2 \cdot (5 \cdot T_{ch} + T_{ex-LOAD} + 2 \cdot T_{ex-IF} + 2 \cdot T_{ex-INCR}) = 450\,\tau\,N^2$$
$$= 450 \cdot \frac{1}{4 \cdot 10^9} \cdot 16^2 \cdot 1024^2 \sim 30{,}2 \cdot 10^9 \cdot 10^{-9} = 30{,}2\,sec$$

Per calcolare la performance a partire da questo tempo di completamento contiamo il numero di istruzioni eseguite nel ciclo della procedura ($5 \cdot N^2$) e calcoliamo l'inverso del tempo medio di elaborazione:

$$\wp = \frac{5 \cdot N^2}{T_c} = 44.4\,MIPS$$

Istruzioni complesse per estendere D-RISC

A partire dall'implementazione del programma della Domanda 1 possiamo definire un certo insieme di istruzioni non D-RISC che possono, in linea di principio, permettere di ottimizzare il tempo di completamento del programma.

Si può notare come le due istruzioni (linee 0 e 1 del programma principale)

LOAD R_A, R_i, R_a
STORE R_{att1}, 0, R_a

consistano nel trasferire un valore presente in una locazione di memoria in un'altra locazione di memoria. È quindi possibile pensare alla definizione di un'istruzione che "muove" il contenuto di una certa locazione di memoria in un'altra locazione di memoria:

MV_MM R_A, R_i, R_{att1} R_{offset}

Quest'istruzione di **M**o**V**e_**M**emoria**M**emoria utilizza un indirizzamento di tipo base + indice tramite il contenuto dei primi due parametri, per ottenere l'indirizzo della locazione "sorgente" e un'indirizzamento base+indice tramite il terzo parametro e il quarto parametro, per ottenere l'indirizzo della locazione "destinazione". La stessa istruzione può essere utilizzata anche per sostituire le due istruzioni (linee 4 e 5 del programma principale)

LOAD R_{att3}, 0, R_c
STORE R_C, R_i, R_c

Mettiamo a confronto il costo dell'esecuzione delle istruzioni alle linee 0 e 1:

$$T_{load} + T_{store} = 2\,T_{ch} + T_{ex-LOAD} + T_{ex-STORE} = 4 \cdot (2\,\tau + t_a) = 296\,\tau$$

$$T_{mv-mm} = T_{ch} + T_{ex-MV\_MM}$$

Dobbiamo quindi analizzare il costo necessario ad eseguire l'istruzione MV_MM. Per semantica dell'istruzione MV_MM si deve trasferire il dato presente in una locazione di memoria in un registro del processore. Il dato è quindi copiato dal registro del processore all'apposita locazione di memoria. Il tempo sarà quindi composto dalle seguenti grandezze:

1. preparazione alla lettura di un dato dalla memoria ($1\,\tau$);
2. lettura in memoria (t_a);
3. preparazione alla scrittura di un dato in memoria ($1\,\tau$);
4. attesa terminazione della scrittura e test delle interruzioni ($1\,\tau$);

Il costo di questa istruzione può essere quindi calcolato come:

$$T_{mv-mm} = 3\,\tau + 2\,t_a = 221\,\tau$$

In questo caso, senza ulteriori ottimizzazioni, l'introduzione di un'istruzione CISC-like può aiutare a diminuire il tempo di completamento del programma.

Un'altra istruzione complessa può sostituire le linee 6 e 7 del programma principale:

INCR R_i
IF < R_i, R_N, LOOP1

e le 7 e 8 della procedura:

INCR R_j
IF < R_j, R_{macro}, LOOP

Chiamiamo tale istruzione IM_INC (**I**f**M**inus_**INC**r) col seguente formato:

IM_INC R_j, R_{macro}, LOOP

Questa istruzione ha la seguente semantica: incrementa il valore contenuto in R_j, controlla che sia minore del valore contenuto in R_{macro} e, se tale condizione è vera, salta all'etichetta LOOP. Anche per questa istruzione possiamo calcolare l'eventuale guadagno in termini di prestazioni. Le due istruzioni D-RISC hanno un tempo di esecuzione pari a:

$$T_{INCR} + T_{IF} = 2 \cdot T_{ch} + T_{ex-INCR} + T_{ex-IF} = 2 \cdot 74\,\tau + \tau + 2\,\tau = 151\,\tau$$

Mentre, dalla semantica "informale" data sopra dell'istruzione IM_INC, possiamo vedere che il suo tempo di esecuzione è pari a:

$$T_{IM\_INC} = T_{ch} + T_{ex-INCR} + T_{ex-IF} = 74\,\tau + \tau + 2\,\tau = 77\,\tau$$

Apparentemente sembra che l'introduzione di questo tipo di istruzione complessa sia di aiuto a diminuire il tempo di completamento di questo programma.

Infine consideriamo le istruzioni alle linee 4 e 5 della procedura:

LOAD R_{arr}, R_j, R_{temp}
IF ≠ R_{val}, R_{temp}, CONT

Le due istruzioni possono essere incapsulate in un'unica istruzione IF_MEM (**IF_MEM**ory) caratterizzata dal seguente prototipo:

IF_MEM R_{arr}, R_j, R_{val}, CONT

In questo caso si noti come l'offset possa occupare al massimo 6 bit: come semantica di questa istruzione, si potrà saltare solo $2^5 - 1$ istruzioni avanti o indietro rispetto alla corrente. Mettiamo a confronto il costo delle due istruzioni precedenti con quella complessa:

$$T_{LOAD} + T_{IF} = 2 \cdot T_{ch} + T_{ex-LOAD} + T_{ex-IF} = 2 \cdot 74\,\tau + 74\,\tau + 2\,\tau = 224\,\tau$$

$$T_{IF\_MEM} = T_{ch} + T_{ex-LOAD} + T_{ex-IF} = 74\,\tau + 74\,\tau + 2\,\tau = 150\,\tau$$

Ottimizzazioni di istruzioni D-RISC

Un'ottimizzazione delle istruzioni D-RISC consiste nell'effettuare gli accessi in memoria per la chiamata delle istruzioni "in parallelo" senza attendere la terminazione di una richiesta precedente. In questo caso possiamo eliminare un T_{ch} da ogni espressione D-RISC.

Ad esempio, il costo necessario ad eseguire le linee 4 e 5 del programma principale

LOAD R_A, R_i, R_a
STORE R_{att1}, 0, R_a

diventa

$$T_{load} + T_{store} = T_{ch} + T_{ex-LOAD} + T_{ex-STORE} = 3 \cdot (2\,\tau + t_a) = 222\,\tau$$

Il guadagno in termini di prestazioni dell'introduzione di un'istruzione complessa è quindi nullo. Stesse considerazioni valgono per le altre due istruzione complesse.

Domanda 3

Il tempo di completamento del programma è espresso come:

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

Il tempo di completamento ideale si ricava dall'espressione del tempo di completamento della Domanda 2 ponendo:

$$t_a = t_c = 3\tau$$

in quanto la cache è completamente associativa. Quindi:

$$T_{c-id} = N^2 \cdot T_{loop} = N^2 \cdot (5 \cdot T_{ch} + T_{ex-LOAD} + 2 \cdot T_{ex-IF} + 2 \cdot T_{ex-INCR}) =$$

$$= N^2 \cdot (5 \cdot 5\tau + 5\tau + 2 \cdot 2\tau + 2 \cdot \tau) = 36\ N^2\ \tau$$

Il tempo medio di trasferimento di un blocco da memoria principale a cache, nel caso in cui la memoria principale sia interallacciata, è pari a:

$$T_{trasf} = 2 \cdot T_{tr} + \frac{\sigma}{m} \cdot \tau_M + m\,\tau$$

dove $m = 4$, $T_{tr} = 10\ \tau$, $\sigma = 8$, $\tau_M = 50\ \tau$. Quindi:

$$T_{trasf} = 124\,\tau$$

Analizziamo il numero di fault generati durante l'esecuzione del programma e determiniamo *l'insieme di lavoro*: quest'ultimo è definito come *l'insieme dei blocchi da mantenere in cache, istante per istante, con l'obiettivo di minimizzare la probabilità di fault*). Per meglio analizzare questi concetti, espandiamo il codice della procedura al posto della sua chiamata nel programma principale:

```
#define K 1024
#define N 16 * K

int main(int argc , char ** argv) {
    int A[N], B[N], C[N];
    int i, j;

    for(i = 0; i < N; i++) {
      for(j = 0; j < N; j++) {
            if(A[i] == B[j]) C[i]++;
      }
    }
   return 0;
}
```

Il codice è soggetto, oltre che a località, a intensivo riuso, ed è contenuto in 4 blocchi (due per il programma principale e due per la procedura). *Questi 4 blocchi di codice fanno parte dell'insieme di lavoro in maniera permanente*. Il loro mantenimento in cache, finché il processo rimane in esecuzione, è assicurato per default dall'implementazione firmware dell'unità cache.

Degli array A e C interessa che, ad ogni iterazione, sia presente in cache il solo blocco corrente, soggetto solo a località. Quindi, *istante per istante fanno parte dell'insieme di lavoro altri 2 blocchi, e precisamente il blocco corrente di A ed il blocco corrente di C*.

L'array B è soggetto tanto a località quanto a *riuso per tutta la durata del programma*. Questa proprietà è verificabile in modo statico osservando le sequenze di generazione degli indici *i* e *j*. Un compilatore ottimizzante, osservando che la dimensione di B è (largamente) minore della capacità della memoria cache, decide di *mantenere in cache tutti i blocchi di B una volta caricativi durante la prima iterazione*. Quindi, *tutti i 2K blocchi di B fanno parte permanentemente dell'insieme di lavoro*. Per imporre questo funzionamento, il compilatore *di questa architettura* genera codice usando l'annotazione *non_deallocare*:

LOOP: LOAD R_{arr}, R_j R_{temp}, **non_deallocare**

Inoltre, nella progettazione del supporto ai processi, verrà imposto che anche altri oggetti, tipicamente il PCB e la Tabella di Rilocazione del processo e la testa/coda della Lista Pronti siano mantenuti permanentemente in cache.

Gli accessi ad A e C provocano un fault ogni σ iterazioni, ma solo i fault sui blocchi di A provocano trasferimento del blocco da memoria principale a cache. Infatti l'unità cache è progettata in maniera da non provocare trasferimento di blocco in seguito a fault in scrittura, ma provvede solo all'allocazione del blocco in cache. Il compilatore ottimizzante, riconoscendo che *l'array C è usato in sola scrittura*, non fa precedere la STORE su C da un'ulteriore istruzione di LOAD su C stesso (istruzione che, con la suddetta progettazione del livello firmware, sarebbe necessaria allo scopo di forzare il trasferimento dei suoi blocchi).

Quindi:

$$N_{fault} = N_{fault\ -A} + N_{fault\ -B} = 2\,\frac{N}{\sigma} = \frac{N}{4}$$

Quindi, la penalità dovuta ai fault di cache:

$$N_{fault} \cdot T_{trasf} = 31\,N\,\tau$$

è *trascurabile* rispetto al tempo di completamento ideale ($36\ N^2\ \tau$). In conclusione:

$$\varepsilon_{cache} \sim 1$$

$$T_c \sim T_{c-id} = 36\ N^2\,\tau = 2{,}4\ sec$$

$$\wp = 555\ MIPS$$

con un miglioramento significativo delle prestazioni, rispetto all'architettura senza cache, di un fattore 450/36 = 12,5.

Appendice A

Questa parte aggiuntiva non fa parte delle risposte ai quesiti posti nella terza esercitazione, ma vuole mostrare la risoluzione di un'altra tipologia di domande relative all'implementazione di istruzioni ASSEMBLER più complesse di quelle che fanno parte del set di istruzioni D-RISC utilizzato nel corso.

Prendiamo in considerazione le tre istruzioni introdotte nella parte finale della soluzione alla Domanda 1 e diamo un'implementazione nel modello di processore del Cap. VI. Per ogni istruzione, mostriamo il suo prototipo, una sua semantica operazionale nello stile di un linguaggio di programmazione ad alto livello e la sua implementazione a microprogramma sul processore: si noti come quest'ultima rappresenti anch'essa una semantica operazionale, ma a più basso livello rispetto alla precedente.

Move Memoria-Memoria

Abbiamo visto che il prototipo di questa istruzione è:

MV_MM R_{base1}, R_{off1}, R_{base2}, R_{off2}

e la sua semantica "informale" corrisponde a:

MV[RG[R_{base1}]+RG[R_{off1}]] → MV[RG[R_{base2}]+RG[R_{off2}]], IC + 1 → IC

L'indirizzamento della memoria virtuale è calcolato come base+indice, rispettivamente per i primi due parametri e per il terzo e il quarto. L'effetto di questa istruzione è quello di trasferire il contenuto della locazione di memoria individuata dal contenuto dei primi due registri nella locazione di memoria individuata dal contenuto del terzo e quarto registro. Mostriamo l'implementazione di questa istruzione come estensione del microprogramma del processore del Cap. VI. Il microcodice parte dall'etichetta immediatamente successiva alla fase di chiamata dell'istruzione:

{esecuzione MV_MM}

mv_mm0. RG[IR.R_{base1}] + RG[IR.R_{off1}] → IND, 'read' → OP, set RDYOUT, mv_mm1;

mv_mm1. (RDYIN, or(ESITO) = 0 -) nop, mv_mm1;

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

(= 1 0) reset RDYIN, RG[IR.R_{base2}] + RG[IR.R_{off2}] → IND, 'write' → OP,

DATAIN → DATAOUT, set RDYOUT, mv_mm2;

mv_mm2. (RDYIN, or(ESITO), INT = 0 - -) nop, mv_mm2;

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

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

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

Incremento e If <

In questa istruzione si incrementa il valore contenuto in un registro (primo parametro) e lo si confronta con quella contenuto in un altro registro (secondo parametro): se il valore del primo è minore del secondo, si esegue un salto di tanti passi rispetto all'istruzione attuale quanti sono indicati nel terzo parametro (offset). Altrimenti si passa, normalmente, all'istruzione successiva.

Il prototipo di questa istruzione è il seguente:

IM_INCR R_{op1}, R_{op2}, OFFSET

e la sua semantica informale è:

RG[R_{op1}] = RG[R_{op1}] + 1; *if* RG[R_{op1}] < RG[R_{op2}] *then* IC = IC + OFFSET *else* IC = IC+1

Analogamente all'istruzione precedente, mostriamo l'implementazione di questa istruzione:

{esecuzione IM_INCR}

im_incr0. RG[IR.R_{op1}] + 1 → RG[IR.R_{op1}], im_incr1;

im_incr1. segno(RG[IR.R_{op1}] – RG[IR.R_{op2}]) → S, zero(RG[IR.R_{op1}] – RG[IR.R_{op2}]) → Z, im_incr2;

im_incr2. (S, Z, INT = 0 0 0) IC + 1 → IC, ch0;

(= 0 0 1) IC + 1 → IC, tratt_int;

(= 1 0 0) IC + IR.offset → IC, ch0;

(= 1 0 1) IC + IR.offset → IC, tratt_int;

If≠ in memoria

Questa istruzione esegue il confronto fra due valori: il primo è contenuto in una locazioni di memoria ed il suo indirizzamento è espresso tramite base+indice (primo e secondo parametro); il secondo è contenuto nel registro generale indicato nel terzo parametro. Se i due valori sono diversi, si salta tanti passi quanti sono indicati dall'offset (terzo parametro); altrimenti si passa, normalmente, all'istruzione successiva.

Il prototipo della funzione è il seguente:

IFD_MEM R_{base}, R_{off}, R_{val}, OFFSET

La sua semantica è:

if MV[RG[R_{base}]+RG[R_{off}]] ≠ RG[R_{val}] *then* IC = IC + OFFSET *else* IC = IC + 1

Vediamo la sua implementazione firmware:

{Esecuzione IFD_MEM}

ifd_mem0. RG[IR.R_{base}]+RG[IR.R_{off}] → IND, 'read' → OP, set RDYOUT, ifd_mem1;

ifd_mem1. (RDYIN, or(ESITO) = 0 -) nop, ifd_mem1;

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

(= 1 0) zero(DATAIN – RG[IR.R_{val}]) → Z, reset RDYIN, ifd_mem2;

ifd_mem2. (Z, INT = 00) IC + 1 → IC, ch0;

(= 01) IC + 1 → IC, tratt_ecc;

(= 10) IC + IR.OFFSET → IC, ch0;

(= 11) IC + IR.OFFSET → IC, tratt_ecc;