Architettura degli elaboratori
Appello 3 settembre 2019 – A.A. 2018—19

Riportare in alto a sinistra di ognuno dei fogli consegnati Nome, Cognome, Matricola e Corso (A o B).
I risultati e la correzione saranno resi noti su didawiki.di.unipi.it appena disponibili, insieme al calendario degli orali.

Domanda 1

Si consideri il microcodice della fase firmware per la gestione delle interruzioni come riportato nel libro di testo e se ne fornisca una versione modificata per l'utilizzo di uno stack per il salvataggio delle informazioni che nella versione originale vengono invece salvate nei registri generali. In particolare, si assuma che in questo caso il registro stack pointer (R_{61}) punti sempre alla prima posizione occupata in testa allo stack e che i registri R_{62} ed R_{63} contengano rispettivamente il limite inferiore (utilizzato per controllare se lo stack é vuoto) e superiore (utilizzato per controllare se lo stack é pieno) dell'aerea di memoria che contiene lo stack e che lo stack cresca da indirizzi bassi verso indirizzi alti.

Successivamente si fornisca il codice della fase assembler del trattamento delle interruzioni, eventualmente modificato per tener conto del diverso metodo di passaggio dei parametri e si spieghi se la modifica é sufficiente a permettere il trattamento di interruzioni annidate. Nelle modifiche apportate al microcodice della fase firmware si tenga come obiettivo il fatto che la nuova versione del trattamento delle interruzioni non comporti un aumento del tempo del ciclo di clock.

Domanda 2

Si consideri lo pseudocodice

```
for(int i=0; i<N; i++)
  for(int j=i; j<N; j++) {
    a[j] = b[i]+c[j];
    c[j]++;
  }
```

e se ne forniscano:

- compilazione in codice D-RISC secondo le regole di compilazione classica
- compilazione ottimizzata del loop centrale considerando (anche) la tecnica del loop unrolling con fattore di unrolling pari a *k*.
- valutazione delle prestazioni, nei due casi, evidenziando gli effetti del loop unrolling.

Per la valutazione delle prestazioni si consideri un processore D-RISC pipeline come quello del libro di testo, senza unità di esecuzione slave, nel quale le addizioni e sottrazioni intere sono svolte dalla EU master.

Domanda 3

Si consideri un processore D-RISC pipeline, superscalare a due vie, dotato della sola EU master (ovvero senza unità EU slave). Si assuma di dover eseguire un certo codice assembler, che esegue su un processore D-RISC pipeline, non superscalare e sempre dotato della sola EU master in un tempo pari a *kt*. Si dica quale potrebbe essere lo speedup ($speedup = T_{pipe} / T_{superscalare}$) massimo e minimo e si forniscano tre esempi di pseudo codice e relativo codice assembler che ottengono rispettivamente uno speedup minimo, uno massimo e uno pari ad un valore intermedio fra il minimo e il massimo.

Traccia di soluzione

Domanda 1

La fase firmware procede restituendo un ACKint all'unità di arbitraggio delle interruzioni, esterna al processore, leggendo due parole dall'interfaccia col sottosistema di memoria e chiamando la fase assembler dopo aver salvato il valore corrente del program counter, da utilizzare per la prosecuzione delle attività dopo che il trattamento di questa interruzione è terminato. Per come lo abbiamo visto a lezione, le due parole (numero dell'unità e tipo di interruzione) e il program counter da salvare vengono memorizzati in registri generali:

```
trattint0. reset INT, set ACKint, trattint1
trattint1. (RDYin, or(ESITO) = 0-) nop, trattint1,
           (=11) ESITO->ESITO1, reset RDYin, trattecc,
           (=10) DATAin->REG[61] set ACKm, reset RDYm, trattint2
trattint2. (RDYm,or(ESITO) = 0-) nop, trattint2,
           (=11) ESITO->ESITO1, reset RDYin, trattecc,
           (=10) DATAin->REG[62], IC->REG[63], REG[60]->IC,
                 set ACKm, reset RDYm, ch0
```

A questa fase fa seguito la fase assembler in cui si usa il numero dell'unità che ha sollevato l'interruzione come indice per accedere ad un vettore di indirizzi delle routine di trattamento delle interruzioni (driver) che vengono chiamate ad interruzioni disabilitate, per poi restituire il controllo all'istruzione successiva a quella che ha verificato la presenza dell'interruzione:

```
LOAD Rvectint, R60, Rdriver, DI
CALL Rret, Rdriver
GOTO R62, EI
```

Per gestire il passaggio dei parametri (numero dell'unità, causa dell'interruzione e indirizzo di ritorno) utilizzando lo stack, possiamo assumere di mettere sullo stack, nell'ordine: indirizzo di ritorno, ragione dell'interruzione e numero dell'unità (così che da poter fare le pop in ordine giusto) e modificare il microcodice come segue:

```
trattint0. // qui comincio a calcolare se ho spazio sullo stack
           // oltre a gestire INT e ACKint
           reset INT, set ACKint, REG[63]-2->LIMITE, trattint1
trattint1. // leggo prima parola e la salvo in TEMP1
           (RDYm, or(ESITO) = 0-) nop, trattint1,
           (=11) ESITO->ESITO1, trattecc,
           (=10) minore (REG[61], LIMITE)->OK,
                 DATAin->TEMP1, set ACKm, reset RDYm, trattint2
trattint2. (RDYm,or(ESITO),OK = 0--) nop, trattint2,
           (=11-) ESITO->ESITO1, trattecc,
           (=100) nop, trattecc,
           // leggo la seconda parola (ragione int)
           // e ordino la prima push sullo stack (ind di ritorno)
           (=101) DATAin-> TEMP2, "write"->OP, REG[61]+1->IND,
                  IC -> DATAout, REG[61]+1 -> REG[61],
```

```
                    reset RDYin, set ACKin, trattint3
trattint3.  (RDYin,or(ESITO)=0-) nop, trattint3,
            (=11) ESITO -> ESITO1, trattecc,
            // ordino la seconda push (ragione dell'int)
            (=10) TEMP2-> DATAout, REG[61]+1 -> IND, "write"->OP,
                  REG[61]+1 -> REG[61],
                  set ACKin, reset RDYin, trattint4
trattint4.  (RDYin,or(ESITO)=0-) nop, trattint3,
            (=11) ESITO -> ESITO1, trattecc,
            // ordino la terza push (numero dell'unità)
            (=10) TEMP1-> DATAout, REG[61]+1 -> IND, "write"->OP,
                  set ACKin, reset RDYin, REG[60] -> IC, ch0
```

A questo punto, il codice della fase assembler potrebbe essere il seguente:

```
// pop numero dell'unità
LOAD R61, #0, Ri, DI
SUB R61, #1, R61
LOAD R60, Ri, Rdriver
// chiamata del driver, che esegue la pop della seconda parola
// (ragione dell'interruzione) dallo stack
CALL Rret, Rdriver
// pop dell'indirizzo di ritorno
LOAD R61, #0, Rret
SUB R61, #1, R61
// ritorno
GOTO Rret, EI
```

La modifica è sufficiente a permettere il trattamento di interruzioni annidate, purchè si eliminino entrambi i flag di disabilitazione e riabilitazione delle interruzioni nella prima e ultima istruzione della fase assembler. Va considerato inoltre che la MMU deve tener conto che al secondo ACK riceverà anche un'operazione da eseguire. Visto che comunque la MMU sta eseguendo un protocollo particolare (non il classico servizio "memoria" per il processore) questa "eccezione" è giustificata e gestibile.

Le modifiche al microcodice non comportano un aumento del ciclo di clock, visto che l'operazione più complessa (REG[…]+…-> IND è la stessa che troviamo anche nella LOAD/STORE e che la condizione di stack pieno è testata utilizzando un registro anziché tramite una variabile di condizionamento complessa.

Soluzione 2

Compilazione secondo le regole:

```
        CLEAR Ri
LOOPi:  MOV Rj, Ri
LOOPj:  LOAD Rb, Ri, R1
        LOAD Rc, Rj, R2
        ADD R1, R2, R3
        STORE Ra, Rj, R3
        INC R2
        STORE Rc, Rj, R2
        INC Rj
        IF< Rj, Rn, LOOPj
        INC Ri
        IF< Ri, Rn, LOOPi
```

Il loop centrale (evidenziato in corsivo e grassetto) ha tre dipendenze: una ADD -> STORE1, una INC –> STORE2 e una INC -> IF<. Avremo dunque tre bolle: una da 2t (vista la presenza delle LOAD nella sequenza di istruzioni che portano alla STORE) per la prima dipendenza, una da 1t per la seconda e infine una da 1t per la terza dipendenza.

A queste bolle ne va aggiunta una da 1t per il salto finale. A fronte di un tempo

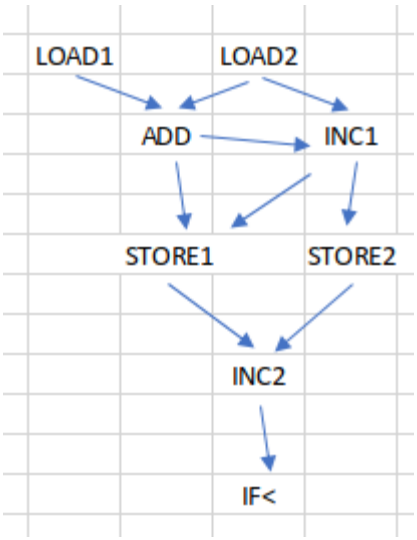
Tid = 8t

abbiamo un

T = 13t

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
IM	LOAD	LOAD	ADD	STORE			INC	STORE		INC	IF<		INC	LOAD					
IU		LOAD	LOAD	ADD	STORE		STORE	INC	STORE	STORE	INC	IF<	IF<	~~INC~~	LOAD				
DM			LOAD	LOAD				STORE			STORE					LOAD			
EU				LOAD	LOAD	ADD			INC			INC					LOAD		

Se consideriamo il grafo delle dipendenze data flow fra le istruzioni assembler



Possiamo notare che si possono invertire l'ordine delle STORE in modo da concludere prima l'aggiornamento del registro che induce la dipendenza.

```
LOOPj: LOAD Rb, Ri, R1
       LOAD Rc, Rj, R2
       ADD R1, R2, R3
       INC R2
       STORE Ra, Rj, R3
       STORE Rc, Rj, R2
       INC Rj
       IF< Rj, Rn, LOOPj
```

In questo caso riusciamo ad ottenere un tempo migliore per la singola iterazione. Rimane infatti una bolla da 1t per la dipendenza indotta dalla ADD sulla STORE (c'è una INC in mezzo ma le LOAD nella sequenza fanno sì che la bolla sia da 2t e quindi 1t di bolla rimane), una bolla da 1t per dipendenza indotta dalla INC Rj sulla IF< e infine una bolla da 1t per il salto. Dunque T = 11t

	0	1	2	3	4	5	6	7	8	9	10	11	12		
IM	LOAD	LOAD	ADD	INC	STORE		STORE	INCj	IF<			LOAD			
IU		LOAD	LOAD	ADD	INC	STOREc	STORE	STORE	INC	IF<	IF<		LOAD		
DM			LOAD	LOAD				STORE	STORE					LOAD	
EU				LOAD	LOAD	ADD	INC			INC					LOAD

Adesso consideriamo l'unrolling, per semplicità con k=2. Il codice del ciclo interno diventa:

```
LOOPj: LOAD Rb, Ri, R1
       LOAD Rc, Rj, R2
       ADD R1, R2, R3
       INC R2
       STORE Ra, Rj, R3
       STORE Rc, Rj, R2
       INC Rj
       IF>= Rj, Rn, LOOPj
       LOAD Rb, Ri, R1
       LOAD Rc, Rj, R2
       ADD R1, R2, R3
       INC R2
```

```
        STORE Ra, Rj, R3
        STORE Rc, Rj, R2
        INC Rj
        IF< Rj, Rn, LOOPj

FINE:   …
```

Rimane il problema delle dipendenze indotte dalle INC Rj sulle IF. Utilizzando un registro base Rx1 decrementato di 1 rispetto a Rx (base del vettore X in memoria virtuale), possiamo anticipare la INC Rj eliminando del tutto l'effetto della dipendenza e contribuendo ad eliminare anche la bolla rimasta per la dipendenza ADD->STORE, utilizzando il codice seguente:

```
LOOPj:  LOAD Rc, Rj, R2
        LOAD Rb, Ri, R1
        ADD R1, R2, R3
        INC R2
        INC Rj
        STORE Ra, Rj, R3
        STORE Rc, Rj, R2
        IF>= Rj, Rn, LOOPj
        LOAD Rc, Rj, R2
        LOAD Rb, Ri, R1
        ADD R1, R2, R3
        INC R2
        INC Rj
        STORE Ra, Rj, R3
        STORE Rc, Rj, R2
        IF< Rj, Rn, LOOPj

FINE:   …
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
IM	LOAD	LOAD	ADD	INC2	INCj	ST2	ST1	IF>=	LOAD	LOAD	ADD	INC2	INCj	ST2	ST1	IF<	XXX	LOAD			
IU		LOAD	LOAD	ADD	INC2	INCj	ST2	ST1	IF>=	LOAD	LOAD	ADD	INC2	INCj	ST2	ST1	IF<	~~XXX~~	LOAD		
DM			LOAD	LOAD				ST2	ST1		LOAD	LOAD				ST2	ST1			LOAD	
EU				LOAD	LOAD	ADD	INC2	INCj				LOAD	LOAD	ADD	INC2	INCj					LOAD

Dunque, rimane, ogni k iterazioni, solo una bolla da salto causata dalla IF< Rj, Rn, LOOPj. Il tempo di completamento è adesso Tid = 8kt, T = 8k + 1. L'efficienza, per k alto, è dunque vicina a quella ideale.

Risposta 3

Il caso pessimo si ha quando tutte le istruzioni del codice, per una delle ragioni che ne impediscono il raggruppamento in un'unica istruzione "lunga", devono essere eseguire in sequenza, utilizzando una NOP per il secondo slot disponibile. Questo accade quando due istruzioni consecutive, candidate per essere raggruppate in un'istruzione lunga:

- Sono in dipendenza fra di loro, oppure
- Sono due istruzioni di salto.

In questo caso lo speedup sarà 1 (nessun guadagno di prestazioni).

Il caso migliore si ha quando le due condizioni precedenti non valgono e tutte le istruzioni lunghe derivanti sono prive di NOP. In questo caso però, lo speedup potrebbe non essere pari a 2, dal momento che le eventuali dipendenze logiche potrebbero essere più pesanti. Se due istruzioni in dipendenza si trovano a distanza *k* sul superscalare si troveranno in generale ad una distanza *k/2* con conseguente aumento dell'impatto della dipendenza.

Un codice del tipo peggiore potrebbe essere quello che deriva dalla compilazione di

```
C[i] = A[B[i+j]];
```

che compilato diventa

```
ADD Ri, Rj, Rs
LOAD RbaseB,Rs,R1
LOAD RbaseA, R1, R2
STORE RbaseC, Ri, R2
```

Notiamo che la ADD induce una dipendenza sulla prima LOAD, questa la induce sulla seconda LOAD e infine la seconda LOAD la induce sulla STORE (rispettivamente a causa dei registri Rs, R1 ed R2, che vengono scritti da ADD, LOAD e LOAD2, e letti da LOAD, LOAD2 e STORE). Questo comporta che il codice superscalare avrà 4 NOP:

```
Loop: ADD Ri, Rj, Rs;        NOP
      LOAD RbaseB,Rs,R1;     NOP
      LOAD RbaseA, R1, R2;   NOP
      STORE RbaseC, Ri, R2;  NOP
```

In questo caso lo speedup è 1, ovvero non vi è alcun miglioramento nel tempo di completamento.

Un codice del tipo "migliore" potrebbe essere il codice centrale dell'algoritmi che calcola risultato e resto della divisione fra interi:

```
while(Resto >= Divisore) {
   Resto -= Divisore;
   Risultato ++;
}
```

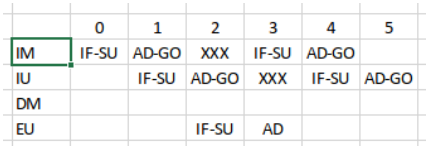
Che si compila nel codice (Resto in R1, Divisore in R2 e Risultato in R3)

```
Loop: IF>= R1, R2, end
      SUB R1, R2, R1
```

```
        ADD R3, #1, R3
        GOTO loop
```

Che a sua volta può essere eseguito sul processore superscalare come

```
Loop:  IF>= R1, R2, end;        SUB R1, R2, R1
       ADD R3, #1, R3;          GOTO loop
```



	0	1	2	3	4	5
IM	IF-SU	AD-GO	XXX	IF-SU	AD-GO	
IU		IF-SU	AD-GO	XXX	IF-SU	AD-GO
DM						
EU			IF-SU	AD		

In questo caso lo speedup è 2 (ovvero il massimo consentito)

Un codice del tipo intermedio potrebbe essere quello che deriva dalla compilazione di

```
for(int i=0; i<n; i++)
  a[i] = b[i];
```

che compilato diventa

```
Loop: LOAD Rb, Ri, R1
      STORE  Ra, Ri, R1
      INC Ri
      IF< Ri, Rn, loop
```

Notiamo che la LOAD induce una dipendenza sulla STORE e la INC sulla IF<, ma la STORE non induce alcuna dipendenza sulla INC. Questo comporta che il codice superscalare avrà 2 NOP:

```
Loop: LOAD Rb, Ri, R1;   NOP
      STORE  Ra, Ri, R1; INC Ri
      IF< Ri, Rn, loop;  NOP
```

In questo caso lo speedup sarà leggermente superiore a 1.