Architettura degli elaboratori
Secondo appello 2018-19 – 5 febbraio 2019

Riportare in altro a destra su ciascuno dei fogli consegnati nome, cognome e numero di matricola. La bozza di correzione e, i risultati e il calendario degli orali saranno resi disponibili via web (didawiki e/o pagina dei docenti) appena disponibili.

Domanda 1

Una unità firmware U contiene una memoria M da 1K parole di 32 bit, in cui è memorizzata una lista di elementi a due campi: il primo (memorizzato in indirizzi pari) contiene un numero intero, il secondo (memorizzato all'indirizzo successivo, quindi dispari), contiene il puntatore all'elemento successivo della lista (se è l'ultimo elemento, allora contiene -1). U implementa due operazioni esterne, "sposta" ed "elimina" che vengono richieste da una unità esterna W secondo un protocollo a domanda risposta.

Per la prima operazione ("sposta") U riceve un valore X da W e ricerca nella lista un elemento il cui primo campo è uguale ad X. Se lo trova, lo sposta in fondo alla lista e restituisce a W "trovato", altrimenti restituisce "non trovato". Per la seconda operazione ("elimina") U riceve da W la richiesta di eliminazione: se la lista non è vuota, elimina il primo elemento della lista e restituisce "eliminato", altrimenti restituisce "non eliminato". Si consideri l'esistenza di due registri (START e END) che indicano inizialmente l'indirizzo del primo e dell'ultimo elemento della lista. I registri vanno mantenuti aggiornati durante lo svolgimento delle operazioni esterne.

Si fornisca un'implementazione di U che minimizza la durata del ciclo di clock, motivando adeguatamente la risposta. Si hanno a disposizione porte logiche a 8 ingressi ed ALU che operano in un tempo $t_{alu} = 6\ t_p$.

Domanda 2

In riferimento al seguente frammento di pseudocodice:

```
for i=n-1, i>0, i--
  { S=0;
    for j=i+1, j<N, j++
       { S=(S+U[i,j]*X[j]) / U[i,i]; }
    X[i]=Y[i] – S;
}
```

si forniscano:

- working set;
- compilazione in assembler D-RISC secondo le regole di compilazione standard;
- prestazioni (tempo di servizio) su un processore D-RISC pipeline con unità EU slave pipeline a 4 stadi per l'esecuzione delle moltiplicazioni e divisioni intere;
- cause di degrado delle prestazioni;
- eventuali ottimizzazioni (quantificando il guadagno in termini di tempo di servizio).

Traccia di soluzione

Domanda 1

L'interfaccia della unità U verso W comprende:

- in ingresso: un RDY, un registro OP da un bit e un registro da 32 bit X
- in uscita: un ESITO da 1 bit (0 -> esito negativo, 1 -> esito positivo), un ACK

Utilizziamo un registro START e un registro END, come suggerito dal testo, per mantenere un puntatore alla testa ed alla coda della lista.
Per minimizzare il ciclo di clock, cerchiamo di non utilizzare variabili di condizionamento complesse e di non utilizzare micro-operazioni che richiedano risorse in cascata (per esempio due ALU in cascata).
Per la prima operazione dovremmo innanzitutto controllare se la lista è vuota o se l'elemento cercato è il primo della lista. Entrambi i casi richiedono un trattamento particolare, che consiste semplicemente nell'invio dell'esito (negativo nel primo caso e positivo nel secondo).
Per la seconda operazione l'unico caso particolare sarà quello di lista vuota, che richiede un esito negativo.
Per l'accesso ad un elemento della lista dovremmo indirizzare due posizioni consecutive della memoria (per esempio START e START+1). Evitiamo l'utilizzo di una ALU per calcolare la START+1 utilizzando, per tale valore, i 31 bit più significativi di START concatenati con un singolo bit a 1, facendo leva sul fatto che sappiamo che l'inizio dell'elemento della lista è sempre a una posizione pari.
Con queste ipotesi, e assumendo di utilizzare ALU per controllare le varie condizioni, possiamo scrivere il microcodice di controllo come segue:

1. (RDY, OP = 0-) nop, 1. // se non ho richieste attendo
   (=10) eq(START, -1) -> V, eq(M[START],X) -> T, 3. // prima op, controllo se vuota e se = 1o elem
   (=11) eq(START, -1) -> V, 2 // seconda op, controllo se lista vuota
2. (V = 0) M[START+1] -> START, 1->ESITO, reset RDY, set ACK, 1. // non vuota, elimino primo elem
   (=1) 0 -> ESITO, reset RDY, set ACK, 1. // vuota, esito negativo
3. (V, T = 1-) 0 -> ESITO, reset RDY, set ACK, 1. // lista vuota, non trovato
   (= 01) M[START+1] -> START, // non vuota, è il primo elemento, aggiorno START a next
   START -> M[END+1], // il primo elemento corrente diventa l'ultimo
   START +1 -> TEMP, 4 // ricordati l'ultimo elmento (evito 2 write cont in M)
   (= 00) START -> PREV, M[START+1] -> CURR, 5. // lista non vuota è il primo elemento
4. -1 -> M[TEMP], 1 -> ESITO, reset RDY, set ACK, 1. // finisco di sistemare l'ultimo elem della lista
5. eq(M[CURR+1], -1) -> V, eq(M[CURR], X) -> T, 6. // controllo se trovato o fine lista
6. (V, T = 1 0) 0 -> ESITO, reset RDY, set ACK, 1. // non trovato, lista finita
   (=00) CURR -> PREV, M[CURR+1] -> CURR, 5. // non trovato, considero il prossimo elem
   (=01) M[CURR+1] -> M[PREV+1], 7. // trovato, lo metto in fondo
7. CURR -> M[END+1], CURR+1 -> TEMP, 4.

Consideriamo di utilizzare una memoria a doppia porta (un selettore per la scrittura e due commutatori per la lettura, quindi in grado leggere due posizioni distinte e scrivere una delle due nello stesso ciclo di clock). Questa caratteristica è sfruttata nalle seconda frase della terza microistruzione (leggiamo la posizione START+1 e scriviamo la posizione END+1), nella quinta microistruzione (leggiamo le posizioni

CURR e CURR+1) e nella terza frase della microistruzione 6 (leggiamo la posizione CURR+1 e scriviamo PREV+1). Occorre dunque anche un commutatore sugli ingressi degli indirizzi, che preveda come ingressi START, START+1, END+1, CURR, CURR+1, PREV, TEMP, distribuiti fra i due ingressi indirizzo disponibili (ingresso 1, che comanda il selettore per il β e uno dei due commutatori di lettura, e ingresso 2 che comanda solo uno dei due commutatori di lettura) in modo da permettere come operazioni concorrenti:

- M[START+1]-> … e … -> M[END+1] (lettura e scrittura)
- M[CURR] -> … e M[CURR+1] -> … (doppia lettura)
- M[CURR+1] -> … e … -> M[PREV+1], (lettura e scrittura)

Il tempo del commutatore va ad aggiungersi al tempo di accesso della memoria (che per 1K posizioni sarebbe 6tp) portando la lettura (o scrittura) della memoria $t_a = 8t_p$.

Il massimo $T_{\sigma PO}$ è un $t_a+t_{alu}+t_k$ (considerando che gli indirizzi dispari siano ottenuti concatenando un bit 1 al posto del bit meno significativo dell'indirizzo pari). Questo accade per esempio nella 5. Possiamo diminuire ancora questo $T_{\sigma PO}$ spezzando gli accessi in memoria e il calcolo con la ALU. Questo può essere ottenuto spezzando la 5 in due istruzioni distinte:

5. M[CURR] -> MCURR, M[CURR+1] -> MCURR1, 5.1.
5.1. eq(MCURR, X) -> T, eq(MCURR1, -1) -> V, 6

La stessa cosa va fatta per la seconda frase della 1. :

1. (RDY, OP = 0-) nop, 1.
   (=10) eq(START, -1) -> V, M[START] -> MSTART, 1.1.
   (=11) eq(START, -1) -> T, 2.

1.1. eq(MSTART,X) -> T, 3

E quindi il microcodice definitivo diventa il seguente:

1. (RDY, OP = 0-) nop, 1. // se non ho richieste attendo
   (=10) eq(START, -1) -> V, M[START]->MSTART, 2. // prima op, controllo se vuota e se = 1o elem
   (=11) eq(START, -1) -> V, 3 // seconda op, controllo se lista vuota
2. eq(MSTART,X) -> T, 4 // solo per evitare ta+talu in cascata
3. (V = 0) M[START+1] -> START, 1->ESITO, reset RDY, set ACK, 1. // non vuota, elimino primo elem
   (=1) 0 -> ESITO, reset RDY, set ACK, 1. // vuota, esito negativo
4. (V, T = 1-) 0 -> ESITO, reset RDY, set ACK, 1. // lista vuota, non trovato
   (= 01) M[START+1] -> START, // non vuota, è il primo elemento, aggiorno START a next
   START -> M[END+1], // il primo elemento corrente diventa l'ultimo
   START +1 -> TEMP, 5 // ricordati l'ultimo elmento (evito 2 write cont in M)
   (= 00) START -> PREV, M[START+1] -> CURR, 6. // lista non vuota è il primo elemento
5. -1 -> M[TEMP], 1 -> ESITO, reset RDY, set ACK, 1. // finisco di sistemare l'ultimo elem della lista
6. M[CURR+1] -> MCURR1, M[CURR] -> MCURR, 7
7. eq(MCURR1, -1) -> V, eq(MCURR, X) -> T, 8. // controllo se trovato o fine lista
8. (V, T = 1 0) 0 -> ESITO, reset RDY, set ACK, 1. // non trovato, lista finita
   (=00) CURR -> PREV, M[CURR+1] -> CURR, 6. // non trovato, considero il prossimo elem
   (=01) M[CURR+1] -> M[PREV+1], 9. // trovato, lo metto in fondo

9. CURR -> M[END+1], CURR+1 -> TEMP, 5. // e finisco di sistemare la fine della lista

In questo modo, per ognuna delle frasi abbiamo al massimo

$T_{\omega PO}$ (cioè 0) + max { $T_{\omega PC}$ + max { $t_{alu}+t_k$, t_a+t_k } , $T_{\sigma PC}$ } + δ

Abbiamo 9 microistruzioni, 4 variabili di condizionamento (2 testate contemporaneamente) e 16 frasi. 4 bit di stato e 2 bit testati come variabili di condizionamento fanno sì che il numero di livelli di porte AND nella ωPC e σPC sia uno solo. Le 16 frasi fanno sì che avremo un massimo di 8 "1" (altrimenti codifichiamo gli zeri e neghiamo l'uscita) e quindi avremo anche un solo livello di porte OR. Il ritardo della ωPC e σPC sarà 2 t_p.

Il ciclo di clock sarà dunque pari a (2 + max {max{6+2, 8+2}, 2} +1) t_p = 13t_p. Meno di questo è impossibile, visto che servono almeno 2t_p per la ωPC (e questo rappresenta il minimo ritardo possibile per ωPC e σPC), 8 t_p per l'accesso in memoria e 2t_p per il commutatore per scrivere nei registri, che hanno input diversi, oltre al t_p per il δ.

Alternativamente, avremmo potuto considerare che il test per controllare se un valore è -1 potrebbe semplicemente controllare il bit più significativo: se è 1 allora il numero è negativo. Siccome i puntatori o sono indirizzi della memoria M (numeri positivi) o –1 (unico valore ammesso negativo), il bit ci dice se il valore è effettivamente –1. Dunque, potremmo aver scritto il codice anche così:

1. (RDY, OP, $START_0$ = 0 - -) nop, 1 // non ho richieste, attesa
   (=1-1) 0 -> ESITO, reset RDY, set ACK, 1. // richiesta con lista vuota, non effettuata
   (=110) M[START+1] -> START, 1 -> ESITO, set ACK, reset RDY, 1. // elimina, lista non vuota
   (=100) eq(M[START],X) -> T, M[START+1]-> NEXT, 2 // controllo primo elemento
2. (T, $NEXT_0$ = 11) 1 -> ESITO, set ACK, reset RDY, 1. // il primo e unico elemento
   (=10) NEXT -> START, -1 -> M[START+1], START->TEMP, 6. // primo elem, non ultimo
   (=01) 0 –> ESITO, set ACK, reset RDY, 1. // unico elemento non trovato
   (=00) START -> PREV, M[START+1] -> CURR, 3. // non trovato, non ultimo elemento, scorro
3. eq(M[CURR],X)- > T, M[CURR+1] -> NEXT, 3 // preparo condizioni da testare
4. (T, $NEXT_0$ = 11) 1 -> ESITO, set ACK, reset RDY, 1. // trovato, è già l'ultimo elemento
   (=10) M[CURR+1] -> M[PREV+1], 5. // trovato, lo salto, aggiorno lista, 1 scrittura alla volta
   (=01) 0 -> ESITO, set ACK, reset RDY, 1. // sono alla fine, non ho trovato, non sposto
   (=00) M[CURR+1] -> CURR, 3. // non trovato, non fine lista, scorro
5. -1 -> M[CURR+1], 7. // aggiusto fine lista
6. TEMP -> M[END+1], set ACK, reset RDY, 1. // finisco di sistemare elem trovato in fondo
7. CURR -> M[END+1], set ACK, reset RDY, 1. // lo metto in fondo

Valgono le stesse considerazioni fatte per il codice precedente. Per esempio, la 3 si può spezzare in una microistruzione che mette M[CURR]->TEMP e una che fa eq(TEMP,X)->T, M[CURR+1]->NEXT in modo da non sommare il tempo di accesso alla memoria al tempo della ALU che calcola eq. Similmente possiamo spezzare la 4a frase della 1. Con 7 microistruzioni (3 bit di stato) e max 2 variabili di condizionamento testate contemporaneamente abbiamo un livello AND nella TωPC e nella TσPC. Il numero di livello OR dipende dal numero delle frasi, che sarebbero 16. Per le stesse considerazioni fatte precedentemente si può assumere che basti un solo livello or nella TωPC e nella TσPC. Dunque TωPC = TσPC = 2t_p. Per la σPO

la frase più lunga sarà della stessa durata di quella che avevamo nel codice precedente. Valgono quindi anche le stesse considerazioni sulla minimalità del τ.

Domanda 2

Working set:

Analizzando lo pseudo codice, vediamo che all'iterazione i accediamo l'ultima parte della riga i-esima della matrice U, il valore di tale riga sulla diagonale (primo elemento dell'ultima parte della riga), l'ultima parte del vettore X nonché la posizione i dei vettori X e Y. Abbiamo località su U, X e Y e riuso su X. Per il codice abbiamo località e riuso, come al solito per codice che contiene cicli. Il working set conterrà dunque una linea con dati di U (quella che contiene un pezzo dell'ultima parte della riga i-esima di U), tutto X (anche se via via la parte iniziale del vettore scompare dal working set) e una linea per Y, oltre al codice.

Compilazione in Assembler D-RISC (in corsivo neretto il loop centrale):

```
        SUB Rn, #1, Ri              // inizializzazione variabile di iterazione i
LOOPi:  ADD R0, R0, Rs              // azzeramento S
        ADD Ri, #1, Rj              // inizializzazione variabile di iterazione j
        MUL Ri, Rn, Rbasei          // calcolo invariante: base riga i rispetto alla base
LOOPj:  ADD Rbasei, Rj, Rind        // offset [i,j]
        LOAD Rbaseu, Rind, R1       // U[i][j]
        LOAD Rbasex, Rj, R2         // X[j]
        MUL R1, R2, R3              // U[i][j] * X[j]
        ADD R3, RS, RS              // S + U[i][j] * X[j]
        ADD Rbasei, Ri, Rind        // offset [i][i]
        LOAD Rbaseu, Rind, Ruii     // U[i][i]
        DIV Rs, Ruii, Rs            // ( S + U[i][j] * X[j]) / U[i][i]
        INC Rj                      // fine ciclo j, incremento variabile di iterazione
        IF< Rj, Rn, LOOPj           // itero se minore di N
        LOAD Rbasey, Ri, Ryi        // Y[i]
        SUB Ryi, Rs, Rtemp          // Y[i] - S
        STORE Rbasex, Ri, Rtemp     // X[i]
        DEC Ri                      // fine ciclo i, decrementa variabile di iterazione
        IF>0 Ri, LOOPi              // itero se maggiore di 0
        END                         // fine lavori
```

Cause di degrado delle prestazioni:

Consideriamo la sola esecuzione del ciclo più interno. Abbiamo una dipendenza IU-EU fra la prima ADD e la LOAD successiva, fra la terza ADD e la LOAD successiva e fra la INC e la IF<. Inoltre, abbiamo una dipendenza EU-EU fra la MUL e la ADD successiva.

Valutazione del tempo di servizio:

Le dipendenze appena elencate portano ad un tempo di completamento dell'iterazione pari a 20t, ovvero ad un tempo di servizio di 2t, come si vede dalla simulazione:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20		
ADD	LOAD		LOAD	MUL	ADD						ADD	LOAD			DIV	INC	IF			ADD		
	ADD	LOAD	LOAD	LOAD	MUL	ADD						ADD	LOAD	LOAD		DIV	INC	IF	IF		ADD	
				LOAD	LOAD										LOAD							
		ADD			LOAD	LOAD	MUL	ADD	ADD	ADD	ADD	ADD	ADD			LOAD	DIV	INC				ADD
								MUL	MUL	MUL	MUL							DIV	DIV	DIV	DIV	

Ottimizzazioni:

Il ciclo più interno si può ottimizzare. La seconda LOAD può essere anticipata fra la prima ADD e la successiva LOAD, cosicché la prima bolla legata alla dipendenza IU-EU sia di fatto eliminata. Il caricamento di U[i][i] può anche essere anticipato per allontanare la ADD dalla MUL che induce la dipendenza EU-EU. Dunque, le due istruzioni (calcolo dell'indice e caricamento della posizione U[i][i] possono essere interposte fra la MUL e la ADD. Fra queste due istruzioni (la ADD induce una dipendenza sulla LOAD) possiamo interporre l'incremento della variabile di iterazione. Infine, la DIV può essere utilizzata nel delay slot della IF di fine iterazione. Il codice così ottimizzato diventa dunque:

```
LOOPj:   ADD   Rbasei,   Rj,   Rind          // offset [i,j]
         LOAD  Rbasex,   Rj,   R2            // X[j]
         LOAD  Rbaseu,   Rind, R1            // U[i][j]
         MUL   R1, R2, R3                    // U[i][j] * X[j]
         ADD   Rbasei,   Ri,   Rind          // offset [i][i]
         INC Rj                              // fine ciclo j, incremento variabile di iterazione
         LOAD  Rbaseu,   Rind, Ruii          // U[i][i]
         ADD   R3, RS, RS                    // S + U[i][j] * X[j]
         IF<   Rj, Rn, LOOPj, delayed        // itero se minore di N
         DIV   Rs, Ruii, Rs                  // ( S + U[i][j] * X[j]) / U[i][i]
```

Possiamo vedere dalla simulazione come l'esecuzione del codice arrivi ad un tempo di servizio quasi ottimale (11t/10):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
ADD	LOAD	LOAD	MUL	ADD	INC	LOAD	ADD	IF<	DIV		ADD	LOAD	LOAD	MUL			
	ADD	LOAD	LOAD	MUL	ADD	INC	LOAD	LOAD	ADD	IF<	DIV	ADD	LOAD	LOAD	MUL		
			LOAD	LOAD					LOAD					LOAD	LOAD		
		ADD		LOAD	LOAD	MUL	ADD	INC		LOAD	ADD	DIV	ADD		LOAD	LOAD	MUL
							MUL	MUL	MUL	MUL			DIV	DIV	DIV	DIV	

se teniamo conto della possibilità di realizzare la comunicazione fra unità in modo asincrono con la quantità di posizioni buffer necessarie.