Politecnico di Milano
Dipartimento di Elettronica e Informazione

prof.ssa Anna Antola
prof. Luca Breveglieri
prof. Giuseppe Pelagatti
prof.ssa Donatella Sciuto
prof.ssa Cristina Silvano

AXO – Architettura dei Calcolatori e Sistemi Operativi

prova di martedì 8 febbraio 2011 – CON SOLUZIONI

Cognome_____ **Nome** _____

Matricola_____ **Firma**_____

Istruzioni

Scrivere solo sui fogli distribuiti. Non separare questi fogli.

È vietato portare all'esame libri, eserciziari, appunti, calcolatrici e telefoni cellulari. Chiunque fosse trovato in possesso di documentazione relativa al corso – anche se non strettamente attinente alle domande proposte – vedrà annullata la propria prova.

Non è permesso lasciare l'aula conservando il tema della prova in corso.

Tempo a disposizione: 2h:15m

Valore indicativo di domande ed esercizi, voti parziali e voto finale:

Esercizio 1 (4 punti) _____

Esercizio 2 (4 punti) _____

Esercizio 3 (4 punti) _____

Esercizi vari (4 punti _____

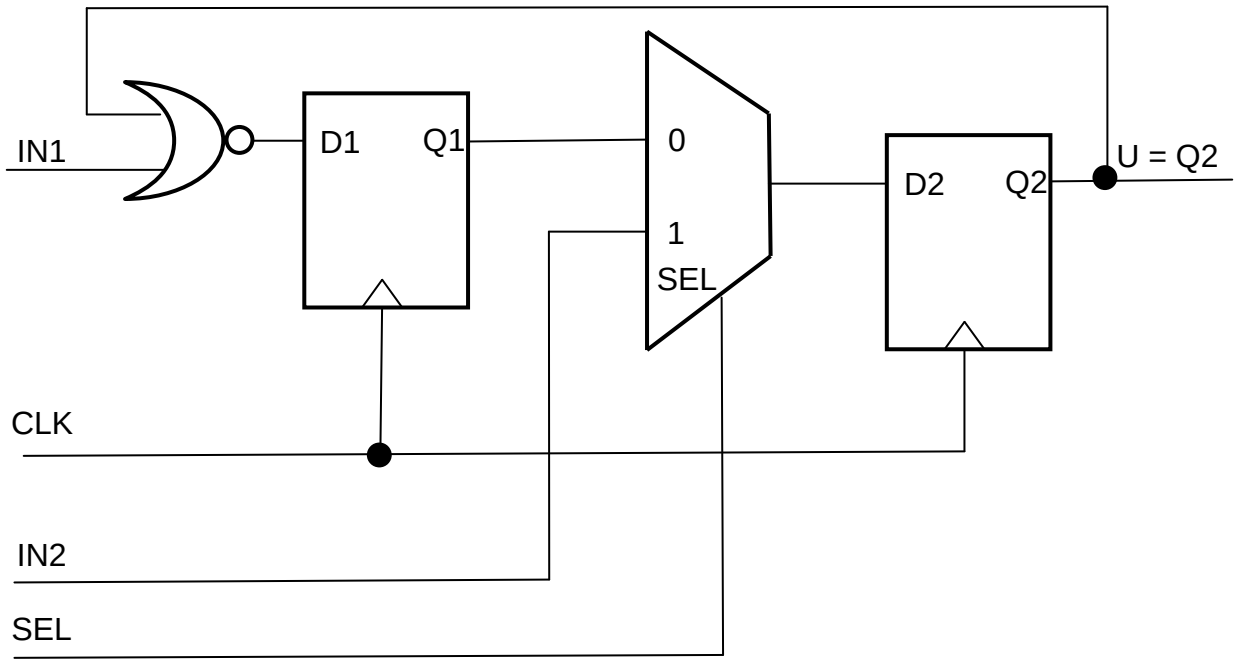
Voto finale: (16 punti) _____

I NUMERI INDICANO I PUNTEGGI APPROSSIMATIVI.

Esercizio n. 1 – logica digitale (4 punti)

prima parte – logica sequenziale

Sia dato il seguente circuito sequenziale:



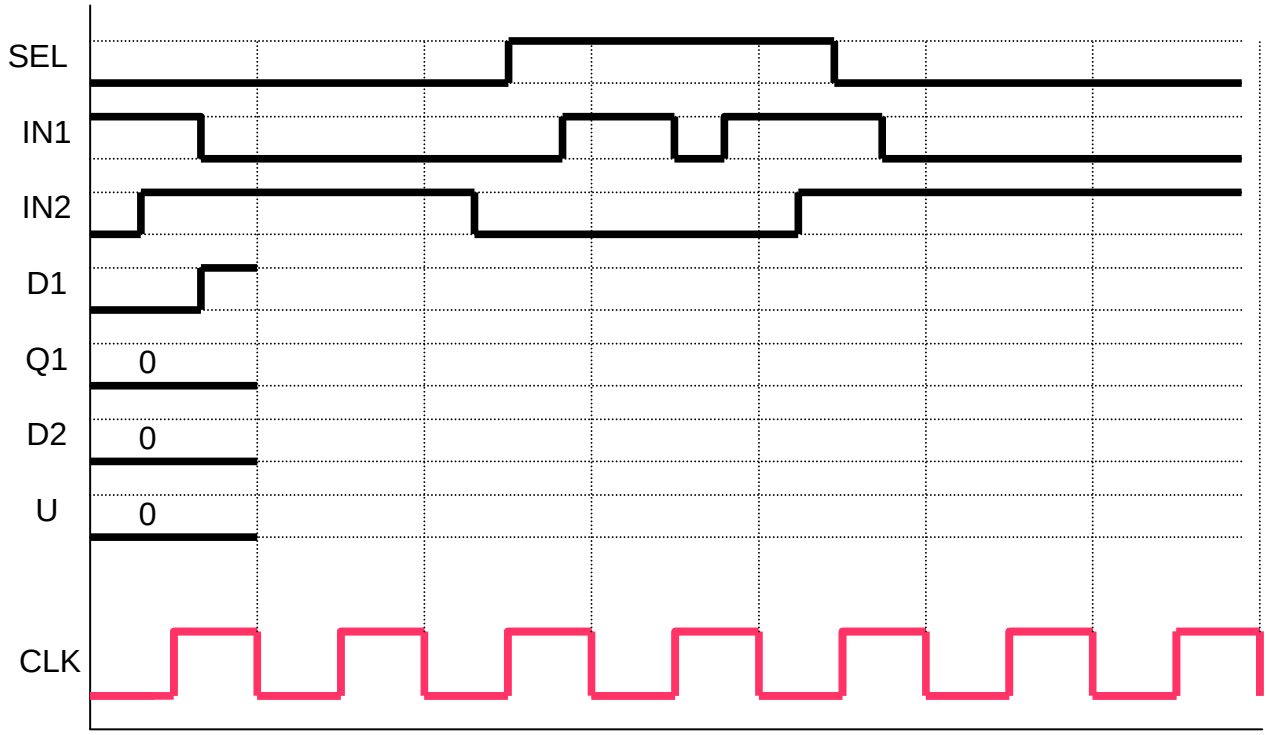
Esso è composto da due bistabili Master / Slave di tipo D: (D1, Q1) e (D2, Q2), con Di ingresso del bistabile e Qi stato / uscita del bistabile; ed è dotato di tre ingressi IN1, IN2 e SEL, e dell'uscita U.

Si fa notare che il valore del segnale D2 è governato da un selettore (multiplexer), che propaga il segnale Q1 o IN2 in funzione del valore del segnale d'ingresso SEL.

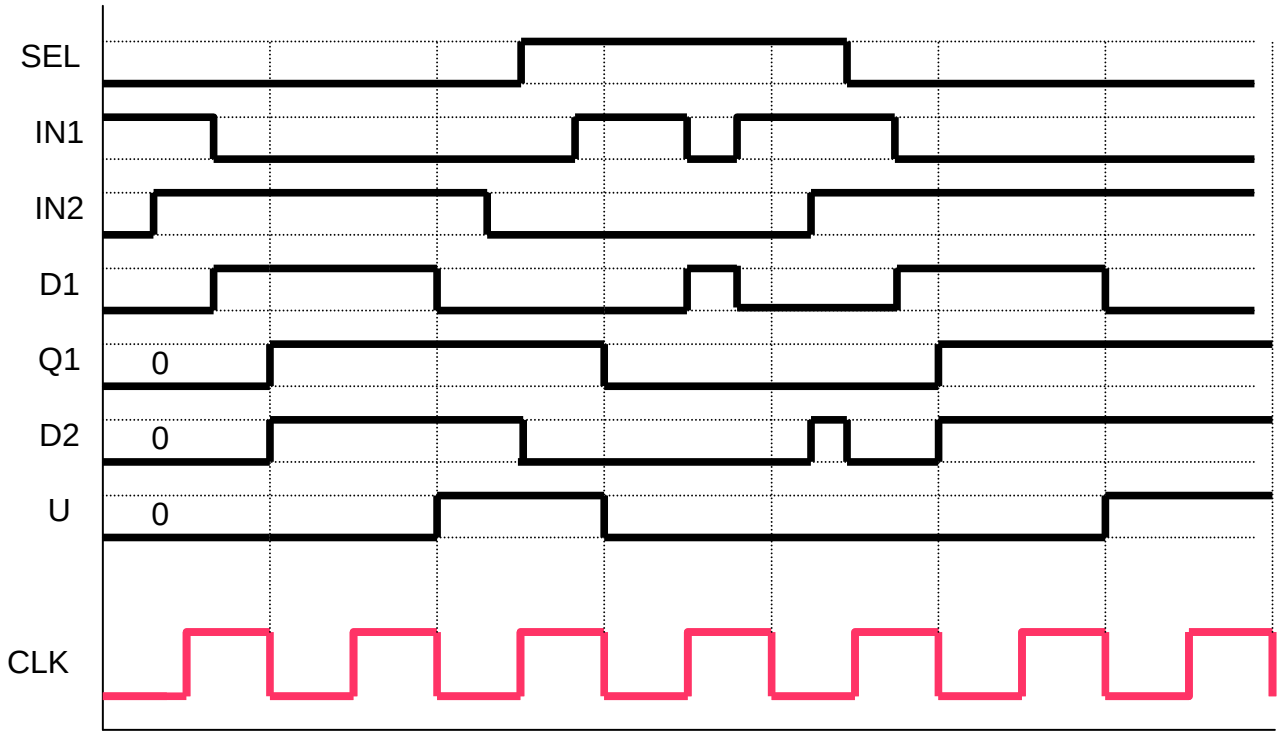
Si chiede di **completare il diagramma temporale** riportato a pagina seguente. Si noti che:

- si devono trascurare completamente i ritardi di propagazione della porta logica NOR e del selettore, e i ritardi di commutazione dei bistabili
- i bistabili sono di tipo "master-slave", la cui uscita commuta sul **fronte di discesa** del clock
- gl'ingressi IN1 e IN2 possono variare in qualunque momento

diagramma temporale da completare



soluzione



seconda parte – logica combinatoria

Si consideri la funzione booleana di quattro variabili G (a, b, c, d) rappresentata dall'espressione seguente:

$$G(a, b, c, d) = !a\ !(b + c) + c\ !(a + b) + !a\ !c\ !(b + d) + c\ !b\ !(a + d)$$

Si trasformi – tramite le proprietà dell'algebra di commutazione – l'espressione di G in modo da ridurre il costo della sua realizzazione, indicando per nome la singola trasformazione svolta oppure la forma della proprietà utilizzata. Allo scopo si usi la tabella seguente (il numero di righe non è significativo).

---

soluzione

espressione trasformata	proprietà
!a !(b + c) + c !(a + b) + !a !c !(b + d) + c !b !(a + d)	*De Morgan*
!a !b !c + c !a !b + !a !c !b !d + c !b !a !d	*commutativa*
!a !b !c + !a !b c + !a !b !c !d + !a !b c !d	*commutativa*
!a !b !c + !a !b !c !d + !a !b c + !a !b c !d	*distributiva*
!a !b !c (1 + !d) + !a !b c (1 + !d)	*elemento nullo*
!a !b !c 1 + !a !b c 1	*elemento unitario*
!a !b !c + !a !b c	*distributiva*
!a !b (!c + c)	*elemento inverso*
!a !b 1	*elemento unitario*
!a !b	*forma minima*

Osservazione; qui i passaggi sono piuttosto dettagliati, ma si possono compattare; in particolare l'uso della proprietà commutativa è alquanto ovvio e si può benissimo lasciare implicito.

esercizio n. 2 – linguaggio macchina (4 punti)

Si deve tradurre in linguaggio macchina simbolico (linguaggio assemblatore) 68000 il programma C (main e funzione *funz*) riportato qui sotto. Nel tradurre non si tenti di accorpare od ottimizzare insieme istruzioni C indipendenti.

La memoria ha **parole** da **16 bit**, **indirizzi** da **32 bit** ed è indirizzabile per byte. Le **variabili intere** sono da **16 bit** e i **puntatori** sono da **32 bit**. Ulteriori specifiche al problema e le convenzioni da adottare nella traduzione sono le seguenti:

- i parametri di tutte le funzioni sono passati sulla pila in ordine inverso di elencazione
- i valori restituiti dalle funzioni ai chiamanti rispettivi sono passati sulla pila, sovrascrivendo il primo dei parametri passati o nello spazio libero opportunamente lasciato
- le variabili locali vengono impilate in ordine di elencazione
- le funzioni devono sempre salvare i registri che utilizzano

Si chiede di:

1. **Riportare**, nelle colonne "domanda 1" delle tabelle, il contenuto **simbolico** (espresso usando i nomi delle variabili, dei parametri, dei registri, ecc) e i **valori dell'indirizzo** dell'area dati statici e dell'area di attivazione della funzione *funz*, così come risultano **subito dopo l'esecuzione dell'istruzione LINK** presente nella funzione stessa.
2. **Riportare**, nella colonne "domanda 2" delle tabelle, il valore (reale, numerico) delle celle **subito prima dell'esecuzione dell'istruzione RETURN durante la terza invocazione della funzione *funz*** (scrivere X se il contenuto è sconosciuto), e il valore numerico contenuto nei **registri FP e SP subito dopo l'esecuzione dell'istruzione LINK**.
3. **Scrivere il codice in linguaggio macchina** 68000 del **main**, coerente con le specifiche e con le risposte ai punti precedenti (il numero di righe non è significativo e le prime righe sono già date).
4. **Scrivere il codice in linguaggio macchina** 68000 della **funzione *funz***, coerente con le specifiche e con le risposte ai punti precedenti (il numero di righe non è significativo e le prime righe sono già parzialmente date).

---

programma in linguaggio C

```
#define N = 4

/* variabili globali                                          */
int vett [N + 1];    /* vettore di 5 interi da 16 bit   */
int i = 0;           /* intero da 16 bit iniz. a 0      */
int r = 0;           /* intero da 16 bit iniz. a 0      */

/* programma main                                             */
void main ( ) {
    /* ciclo inizializz. vettore (non codificato)       */
    /* alla fine vettore contiene valori: 3, 5, 6, 8, 1 */
    while (i < N) do {
      if (vett [i] < vett [i + 1]) {
            r = funz (vett [i], i);
      } /* if */
      i++;
   } /* while */
} /* main */

/* funzione f                                                 */
int funz (int a, int b) {
   /* variabili locali                                        */
   int * x;          /* puntatore da 32 bit                */
   int y;            /* intero da 16 bit                   */
   /* parte esecutiva                                         */
   x = &a;           /* assegna indirizzo di “a” a “x”  */
   y = b;
   return y;
} /* funzione f */
```

spazio per le variabili globali

indirizzo	**contenuto** (simbolico) (domanda 1)	
...	**celle già occupate**	
10000	*vet [0]*	*- 2 byte*
10002	*vet [1]*	*- 2 byte*
10004	*vet [2]*	*- 2 byte*
10006	*vet [3]*	*- 2 byte*
10008	*vet [4]*	*- 2 byte*
10010	*i*	*- 2 byte*
10012	*r*	*- 2 byte*
10014		
...		

area di attivazione della <u>funzione *funz*</u>

indirizzo	**contenuto** (simbolico) (domanda 1)		**valore** (reale) **subito prima dell'esecuzione dell'istruzione RETURN durante la terza invocazione di funz** (domanda 2)
100022	*Y*	*- 2 byte*	*2*
100024	*X*	*- 4 byte*	*100036 (indirizzo di param a)*
100028	*FP precedente*	*- 4 byte*	*X*
100032	*indirizzo di rientro*	*- 4 byte*	*X*
100036	*A*	*- 2 byte*	*6*
100038	*B (e valore uscita)*	*- 2 byte*	*2*
100040	**cella già occupata**		

	valore (reale) **subito dopo l'esecuzione dell'istruzione LINK** (domanda 2)
registro FP	*100028*
registro SP	*100022*

codice in linguaggio macchina 68000 del *main* (domanda 3) – **VARIANTE 1a**			
	ORG	10000	// indirizzo virtuale del segmento unico
N:	**EQU**	4	// costante N = 4
VETT:	**DS.W**	N + 1	// vettore di N + 1 = 5 interi da 16 bit
I:	**DC.W**	0	// varglob i a 16 bit inizializzata a 0
R:	**DC.W**	0	// varglob r a 16 bit inizializzata a 0
MAIN:	**MOVEA.L**	#VETT, A0	// carica costante VETT in A0 (32 bit)
WHILE:	**MOVE.W**	I, D0	// carica varglob i in D0
	CMPI.W	#N, D0	// confronta D0 con costante N = 4
	BGE	END_WHI	// se esito >= 0 va' a END_WHI
IF:	**MOVE.W**	(A0), D1	// carica elemento VETT [i] in D1 **(a)**
	ADDA.L	#2, A0	// somma costante 2 ad A0 (32 bit) **(b)**
	MOVE.W	(A0), D2	// carica elemento VETT [i + 1] in D2
	CMP.W	D2, D1	// confronta D1 con D2
	BGE	END_IF	// se esito >= 0 va' a END_IF
THEN:	**MOVE.W**	I, -(SP)	// impila param b di funz
	MOVE.W	D1, -(SP)	// impila param a di funz (OTTIMIZZATO)
	BSR	FUNZ	// chiama funz
	ADDA.L	#2, SP	// abbandona param a di funz (32 bit)
	MOVE.W	(SP)+, R	// spila uscita di funz in varglob r
END_IF:	**MOVE.W**	I, D0	// carica varglob i in D0
	ADDI.W	#1, D0	// somma costante 1 a D0
	MOVE.W	D0, I	// memorizza D0 in varglob i
	BRA	WHILE	// torna a WHILE
END_WHI:	**END**	MAIN	// fine main
// le istruzioni (a) e (b) sono compattabili tramite autoincremento			
IF:	**MOVE.W**	(A0)+, D1	// carica elem VETT [i] in D1 e inc. A0

Osservazioni: le operazioni sui dati sono a 16 bit e i registri di tipo D sono gestiti a 16 bit (suffisso W); le operazioni sugli indirizzi sono a 32 bit e i registri di tipo A sono gestiti a 32 bit (suffisso L); l'accesso al vettore VETT è realizzato tramite indice; l'indirizzo iniziale di VETT viene caricato nel registro A0 e questo poi funziona come indice, incrementandolo di 2 poiché l'elemento del vettore è da 16 bit ossia 2 byte; questa soluzione è molto semplice, tuttavia dipende dal fatto che la variabile intera i usata per scandire il vettore ha valore iniziale costante uguale a 0; qui comunque è una soluzione valida.

codice in linguaggio macchina 68000 del *main* (domanda 3) – **VARIANTE 1b**			
	ORG	10000	// indirizzo virtuale del segmento unico
N:	**EQU**	4	// costante N = 4
VETT:	**DS.W**	N + 1	// vettore di N + 1 = 5 interi da 16 bit
I:	**DC.W**	0	// varglob i a 16 bit inizializzata a 0
R:	**DC.W**	0	// varglob r a 16 bit inizializzata a 0
MAIN:	**MOVEA.L**	#VETT, A0	// carica costante VETT in A0 (32 bit)
WHILE:	**CMPI.W**	#N, I	// confronta varglob i con cost. N = 4
	BGE	END_WHI	// se esito >= 0 va' a END_WHI
IF:	**MOVE.W**	(A0)+, D1	// carica elem. VETT [i] in D1 e inc. A0
	CMP.W	D1, (A0)	// confronta VETT [i] e VETT [i + 1]
	BLE	END_IF	// se esito <= 0 va' a END_IF
THEN:	**MOVE.W**	I, -(SP)	// impila param b di funz
	MOVE.W	D1, -(SP)	// impila param a di funz
	BSR	FUNZ	// chiama funz
	ADDA.L	#2, SP	// abbandona param a di funz (32 bit)
	MOVE.W	(SP)+, R	// spila uscita di funz in varglob r
END_IF:	**ADDI.W**	#1, I	// somma costante 1 a varglob i
	BRA	WHILE	// torna a WHILE
END_WHI:	**END**	MAIN	// fine main

Osservazioni: questa soluzione è analoga alla VARIANTE 1a, ma è completamente ottimizzata evitando di ricaricare variabili già presenti nei registri, sfruttando l'ortogonalità o la semiortogonalità delle istruzioni e ricorrendo ai modi d'indirizzamento disponibili come risultano dal manuale assembler 68000 (si veda il testo del corso "Introduzione all'architettura dei calcolatori" – McGraw-Hill); in particolare il vettore è scandito tramite auto-incremento dell'indice; la struttura del ciclo a condizione iniziale è mantenuta; verosimilmente questa è tra le soluzioni con listato più breve e qui è mostrata per completezza, poiché non si richiede di ottimizzare bensì di produrre una soluzione corretta.

codice in linguaggio macchina 68000 del *main* (domanda 3) – **VARIANTE 2**			
	ORG	10000	// indirizzo virtuale del segmento unico
N:	**EQU**	4	// costante N = 4
VETT:	**DS.W**	N + 1	// vettore di N + 1 = 5 interi da 16 bit
I:	**DC.W**	0	// varglob i a 16 bit inizializzata a 0
R:	**DC.W**	0	// varglob r a 16 bit inizializzata a 0
MAIN:	**MOVEA.L**	#0, A0	// carica costante 0 in A0 (32 bit)
WHILE:	**MOVE.W**	I, D0	// carica varglob i in D0
	CMPI.W	#N.W, D0	// confronta D0 con costante N = 4
	BGE	END_WHI	// se esito >= 0 va' a END_WHI
IF:	**MOVE.W**	#VETT(A0), D1	// carica elemento VETT [i] in D1
	ADDA.L	#2, A0	// somma costante 2 ad A0 (32 bit)
	MOVE.W	#VETT(A0), D2	// carica elemento VETT [i + 1] in D2
	CMP.W	D2, D1	// confronta D1 con D2
	BGE	END_IF	// se esito >= 0 va' a END_IF
THEN:	**MOVE.W**	I, -(SP)	// impila param b di funz
	MOVE.W	D1, -(SP)	// impila param a di funz (OTTIMIZZATO)
	BSR	FUNZ	// chiama funz
	ADDA.L	#2, SP	// abbandona param a di funz (32 bit)
	MOVE.W	(SP)+, R	// spila uscita di funz in varglob r
END_IF:	**MOVE.W**	I, D0	// carica varglob i in D0
	ADDI.W	#1, D0	// somma costante 1 a D0
	MOVE.W	D0, I	// memorizza D0 in varglob i
	BRA	WHILE	// torna a WHILE
END_WHI:	**END**	MAIN	// fine main

Osservazioni: le operazioni sui dati sono a 16 bit e i registri di tipo D sono gestiti a 16 bit (suffisso W); le operazioni sugli indirizzi sono a 32 bit e i registri di tipo A sono gestiti a 32 bit (suffisso L); l'accesso al vettore VETT è realizzato tramite spiazzamento e indice; l'indirizzo iniziale di VETT è reso come spiazzamento e il registro A0 funziona come indice, incrementandolo di 2 unità poiché l'elemento del vettore è da 16 bit ossia 2 byte; ciò impone che l'indirizzo iniziale del vettore non richieda più di 16 bit, ossia quanti ne sono disponibili per codificare lo spiazzamento numericamente; questa soluzione è più flessibile della precedente poiché l'indice iniziale da caricare in A0 può essere una costante diversa da zero o una variabile.

codice in linguaggio macchina 68000 del main (domanda 3) – **VARIANTE 3**			
	ORG	10000	// indirizzo virtuale del segmento unico
N:	**EQU**	4	// costante N = 4
VETT:	**DS.W**	N + 1	// vettore di N + 1 = 5 interi da 16 bit
I:	**DC.W**	0	// varglob i a 16 bit inizializzata a 0
R:	**DC.W**	0	// varglob r a 16 bit inizializzata a 0
MAIN:	**MOVEA.L**	#VETT, A0	// carica costante VETT in A0 (32 bit)
WHILE:	**MOVE.W**	I, D0	// carica varglob i in D0
	CMPI.W	#N, D0	// confronta D0 con costante N = 4
	BGE	END_WHI	// se esito >= 0 va' a END_WHI
IF:	**MOVE.W**	I, D0	// carica varglob i in D0
	MULS	#2, D0	// raddoppia D0 (MULS è a 16 bit)
	MOVE.W	(A0, D0), D1	// carica elemento VETT [i] in D1
	MOVE.W	I, D0	// carica varglob i in D0
	ADDI.W	#1, D0	// somma costante 1 a D0
	MULS	#2, D0	// raddoppia D0 (MULS è a 16 bit)
	MOVE.W	(A0, D0), D2	// carica elemento VETT [i + 1] in D2
	CMP.W	D2, D1	// confronta D1 con D2
	BGE	END_IF	// se esito >= 0 va' a END_IF
THEN:	**MOVE.W**	I, -(SP)	// impila param b di funz
	MOVE.W	D1, -(SP)	// impila param a di funz (OTTIMIZZATO)
	BSR	FUNZ	// chiama funz
	ADDA.L	#2, SP	// abbandona param a di funz (32 bit)
	MOVE.W	(SP)+, R	// spila uscita di funz in varglob r
END_IF:	**MOVE.W**	I, D0	// carica varglob i in D0
	ADDI.W	#1, D0	// somma costante 1 a D0
	MOVE.W	D0, I	// memorizza D0 in varglob i
	BRA	WHILE	// torna a WHILE
END_WHI:	**END**	MAIN	// fine main

Osservazioni: le operazioni sui dati sono a 16 bit e i registri di tipo D sono gestiti a 16 bit (suffisso W); le operazioni sugli indirizzi sono a 32 bit e i registri di tipo A sono gestiti a 32 bit (suffisso L); l'accesso al vettore VETT è realizzato tramite base A0 e indice D0; naturalmente l'indice D0 va raddoppiato per allinearlo agl'indirizzi pari poiché l'elemento del vettore è da 16 bit cioè 2 byte; questa soluzione è assai flessibile ossia senza limitazioni particolari né di base né d'indice, però è un po' articolata.

codice in linguaggio macchina 68000 della funzione *funz* (domanda 4) **– VAR. 1a**			
FUNZ:	**LINK.L**	FP, #-6	// area di attivazione di funz
B:	**EQU**	+10	// spiazzamento param b
A:	**EQU**	+8	// spiazzamento param a
X:	**EQU**	-4	// spiazzamento varloc x
Y:	**EQU**	-6	// spiazzamento varloc y
	MOVEM.L	A0, -(SP)	// salva registri (32 bit)
(a)	**MOVEA.L**	FP, A0	// copia FP in A0 (32 bit)
(b)	**ADDA.L**	#A, A0	// somma costante A ad A0 (32 bit)
	MOVEA.L	A0, X(FP)	// memorizza A0 in varloc x (32 bit)
	MOVE.W	B(FP), Y(FP)	// copia param b in varloc y (16 bit)
(c)	**MOVE.W**	Y(FP), B(FP)	// sovrascrivi uscita in pila (16 bit)
	MOVEM.L	(SP)+, A0	// ripristina registri (32 bit)
	UNLK.L	FP	// area di attivazione di funz
	RTS		// rientra a chiamante
// si possono compattare le istruzioni (a) e (b) usando l'istr. LEA			
// LEA carica in A0 l'indirizzo del parametro a e non il valore di a			
(a, b)	**LEA.L**	A(FP), A0	// A0 ← A + [FP] (32 bit)
// l'istruzione (c) è sopprimibile poiché qui non modifica la varloc b			
// usualmente però serve poiché inserisce il valore di uscita in pila			

Osservazioni: le operazioni sui dati sono a 16 bit e i registri di tipo D sono gestiti a 16 bit (suffisso W); le operazioni sugli indirizzi sono a 32 bit e i registri di tipo A sono gestiti a 32 bit (suffisso L); volendo si può usare l'istruzione LEA che calcola l'indirizzo del parametro "a" e lo scrive in A0, senza caricare il valore di "a".

Osservazioni generali (per main e funz): onde mantenerlo ben leggibile, il codice macchina è poco ottimizzato, salvo mantenere l'elemento VETT [i] in D1 senza ricaricarlo per passarlo come parametro a funz, e qua e là sfruttare l'ortogonalità di MOVE; sono ottimizzazioni ragionevoli e pressoché immediate; varie altre ottimizzazioni sono possibili, per esempio usando istruzioni specializzate, e qualcuna è indicata in calce.

codice in linguaggio macchina 68000 della funzione *funz* (domanda 4) **– VAR. 1b**			
FUNZ:	**LINK.L**	FP, #-6	// area di attivazione di funz
B:	**EQU**	+10	// spiazzamento param b
A:	**EQU**	+8	// spiazzamento param a
X:	**EQU**	-4	// spiazzamento varloc x
Y:	**EQU**	-6	// spiazzamento varloc y
	MOVEM.L	A0, -(SP)	// salva registri (32 bit)
	LEA.L	A(FP), A0	// carica ind. varloc a in A0 (32 bit)
	MOVEA.L	A0, X(FP)	// memorizza A0 in varloc x (32 bit)
	MOVE.W	B(FP), Y(FP)	// copia param b in varloc y (16 bit)
	MOVEM.L	(SP)+, A0	// ripristina registri (32 bit)
	UNLK.L	FP	// area di attivazione di funz
	RTS		// rientra a chiamante

Osservazioni: questa è la versione ottimizzata di funz, come suggerito nella versione 1a precedente.

esercizio n. 3 – microarchitettura (4 punti)

Lo schema a destra illustra l'architettura di processore a un solo bus interno.

Si consideri l'istruzione macchina seguente del processore 68000:

MOVE (A0), spi (A1, D1)

la cui interpretazione in RTL è:

[A1] + [D1] + spi ← [[A0]]

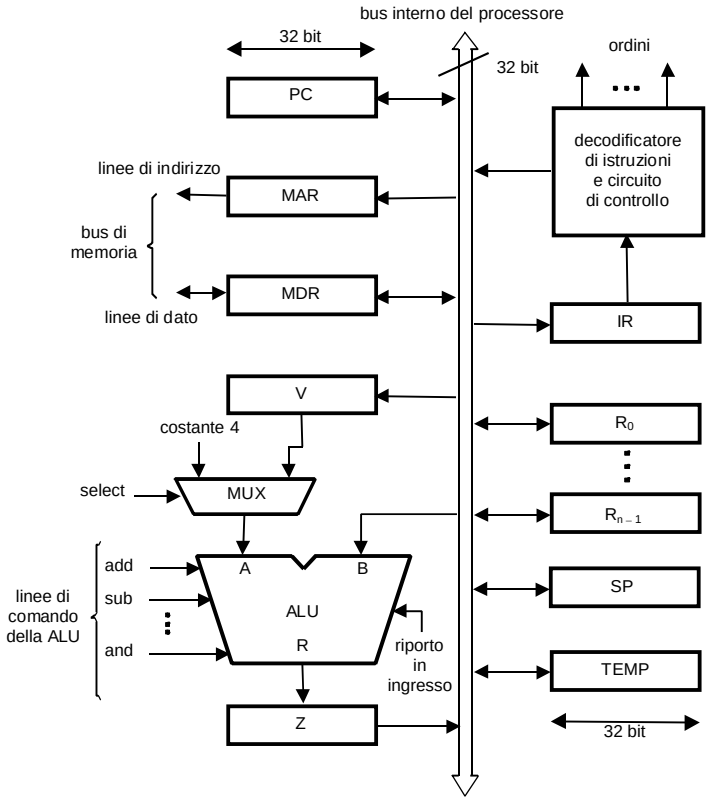
L'istruzione copia il contenuto della parola di memoria puntata dal registro **A0**, nella parola di memoria il cui indirizzo si ottiene sommando il contenuto del registro **A1** con quello del registro **D1** e con lo spiazzamento **spi**. Dati, indirizzi e spiazzamento sono tutti da 32 bit.

L'istruzione ingombra due parole da 32 bit in totale: una parola di codice operativo e una parola aggiuntiva per codificare lo spiazzamento **spi**.

I registri A0, A1 e D1 sono di uso generale (general purpose) e nei passi di controllo vanno indicati con i nomi A0, A1 e D1 (comandi $A0_{in}$, $A0_{out}$, ecc).

Il processore ha cache istruzioni e dati separate: l'ordine **fetch** legge da cache istruzioni, e gli ordini **read** e **write** leggono e scrivono da e in cache dati.

Il tempo di accesso a cache istruzioni vale **tre cicli di clock**, compreso quello dove si dà l'ordine fetch; i tempi di accesso a cache dati sono **ignoti**.



L'unità di controllo è di tipo cablato.

1) Si scriva la sequenza di passi di controllo dell'istruzione MOVE illustrata sopra.

Si usi la tabella a pagina seguente. Nello scrivere la sequenza di passi di controllo, si cerchi di ottimizzarla quanto possibile sfruttando i tempi di accesso a memoria per svolgere attività utili durante le operazioni di memoria. Si ricordi che durante un'operazione di memoria non si possono modificare i registri MAR e MDR.

2) Analisi del comportamento della sequenza progettata.

Il parametro **tempo di accesso** indica il numero di cicli di clock necessari affinché la **cache dati** esegua l'accesso (in lettura o scrittura) ossia indica quanti cicli, compreso quello dove si dà l'ordine **read** o **write**, trascorrono affinché il segnale MFC arrivi al processore. Per esempio:

se tempo di accesso = 1 l'operazione di memoria termina nello stesso ciclo dell'ordine

se tempo di accesso = 2 l'operazione di memoria termina alla fine del ciclo successivo all'ordine

e così via ...

Si calcoli quanti **cicli di clock** occorrono per prelevare ed eseguire l'istruzione MOVE in base alla sequenza di controllo progettata prima, secondo le diverse ipotesi relative al tempo di accesso a cache dati indicate sulle colonne della tabella sotto. L'istruzione è da considerare completata solo quando parte il primo passo di prelievo dell'istruzione successiva. Indipendentemente dal tempo di accesso a cache dati, la cache istruzioni esegue l'accesso in tre cicli compreso quello dove si dà l'ordine fetch, ossia ha sempre tempo di accesso = 3.

tempo di accesso (cache dati)	**1**	**2**	**3**	**4**
numero totale di cicli per prelevare ed eseguire MOVE	*12*	*13*	*14*	*16*

si compili la tabella seguente (il numero di righe non è significativo)

passo	ordini	descrizione RTL
1	PC_{out}, MAR_{in}, *fetch*, *select* 4, *add*, Z_{in}	MAR ← [PC], Z ← [PC] + 4, *leggi cache istr.*
2	Z_{out}, PC_{in}, V_{in}, WMFC (*fetch*)	PC, V ← [Z], *attendi cache istr.*
3	MDR_{out}, IR_{in}	IR ← [MDR]
4	PC_{out}, MAR_{in}, *fetch*, *select* 4, *add*, Z_{in}	MAR ← [PC], Z ← [PC] + 4, *leggi cache istr.*
5	Z_{out}, PC_{in}	PC ← [Z]
6	$A1_{out}$, V_{in}, WMFC (*fetch*)	V ← [A1], *attendi cache istr.* (omissibile)
7	MDR_{out}, *select* V, *add*, Z_{in}	Z ← [V] + [MDR]
8	$A0_{out}$, MAR_{in}, *read*	MAR ← [A0], *leggi cache dati*
9	$D1_{out}$, V_{in}	V ← [D1]
10	Z_{out}, *select* V, *add*, Z_{in}, WMFC (*read*)	Z ← [V] + [Z], *attendi cache dati*
11	Z_{out}, MAR_{in}, *write*, WMFC (*write*), *end*	MAR ← [Z], *scrivi cache dati e attendi, fine*
12		
13		
14		
15		
16		
17		
18		

Osservazione su (1): sono possibili varie soluzioni ma, dato che i tempi di accesso a cache dati sono ignoti, nei limiti del possibile conviene distanziare gli ordini read e write dagli ordini WMFC corrispondenti, e più particolarmente tra read e WMFC giacché write andrà per forza impartito alla fine; dunque importa mettere il più possibile in parallelo la lettura della parola di memoria puntata da A0 e il calcolo dell'indirizzo destinazione spi + [A1] + [D1]; trascorsi due cicli dal prelievo della parola aggiuntiva d'istruzione (passo 6), si libera MDR immediatamente (passo 7) e si fa subito partire l'operazione di lettura della parola di memoria puntata da A0 (passo 8), mentre prosegue il calcolo dell'indirizzo destinazione; i passi 6 e 9 sono interscambiabili poiché in definitiva i registri A1 e D1 non hanno caratteristiche distintive e ovviamente le due addizioni nel calcolo dell'indirizzo destinazione sono commutabili; tuttavia ogni altra sequenza di controllo che non liberi MDR dopo due cicli dal prelievo della parola aggiuntiva d'istruzione, è necessariamente penalizzante poiché riduce il numero di passi intercorrenti tra la lettura e la scrittura della cache dati.

Osservazione su (2): il microcodice consta di 11 passi; dunque la durata dell'istruzione MOVE non è inferiore a 11 cicli di clock; tuttavia WMFC di fetch al passo 2 aggiunge un ciclo al passo dove figura, mentre WMFC di fetch al passo 6 non ha effetto poiché dista già due passi dall'ordine fetch (anzi qui WMFC sarebbe sopprimibile); dunque se la cache dati non influisce (ossia se tempo = 1) la durata minima è di 12 cicli; ora considerando il tempo di cache dati: WMFC di write al passo 11 aggiunge un ciclo per tempo – 1; WMFC di read al passo 10 non ha effetto quando tempo ≤ 3 poiché dista già due passi dall'ordine read, mentre quando tempo = 4 aggiunge un ciclo al passo dove figura; ora è facile calcolare la durata di MOVE: partendo da 12 cicli si va in progressione fintantoché tempo ≤ 3, poi s'inserisce l'attesa di WMFC al passo 10.

esercizio n. 4 – argomenti vari (4 punti)

memoria cache

Si consideri una gerarchia di memoria composta da memoria centrale da **4 Gigabyte** indirizzabile per byte con parole da 32 bit, una memoria **cache istruzioni** a **indirizzamento diretto** (direct mapped) da **512 Kilobyte** e una memoria **cache dati set associativa** a **2 vie** da **1 Megabyte**, entrambe con blocchi da **256 byte**. Il tempo di accesso alla parola di memoria cache (dato o istruzione) è pari a **1 ciclo di clock**. Il tempo di accesso alla parola di memoria centrale è pari a **10 cicli di clock**.

Si chiede di svolgere i punti seguenti:

1. Indicare la struttura degli indirizzi di memoria per le due memorie cache.
2. Calcolare il tempo necessario per caricare un blocco in caso di fallimento (miss).
3. Calcolare il tempo medio di accesso alla memoria per un programma dove in media il **25 %** delle istruzioni eseguite richiede un accesso a dato in lettura o scrittura. Il miss rate (frequenza di fallimento) della cache istruzioni è pari a **1 %** mentre quello della cache dati è pari a **10 %**.
4. Calcolare il rapporto di prestazioni dell'architettura senza cache rispetto a quella con cache.

---

soluzione

Punto 1:

memoria centrale:

32 bit di indirizzo	*4 Giga byte = 2^{32} byte*
8 bit per byte nel blocco	*256 byte = 2^{8} byte*

cache istruzioni:

11 bit per blocco in cache	*512 Kilo byte / 256 byte = 2^{11} blocchi*
13 bit di etichetta	*32 bit − 11 bit − 8 bit = 13 bit*

cache dati:

11 bit per gruppo (set) in cache	*1 Mega byte / (256 byte × 2 vie) = 2^{11} gruppi*
13 bit di etichetta	*32 bit − 11 bit − 8 bit = 13 bit*

Punto 2:

n. di parole per blocco = 256 byte / 4 byte per parola = 64 parole

penalità di fallimento (miss penalty) = 10 cicli di clock × 64 parole = 640 cicli di clock (ev. + 1)

Punto 3:

T medio di accesso senza cache = 10 cicli di clock

T medio istruzioni = 0,99 hit × 1 ciclo + 0,01 miss × (640 + 1) penalità = 7,4 cicli di clock

T medio dati = 0,9 hit × 1 ciclo + 0,1 miss × (640 + 1) penalità = 65 cicli di clock

T medio di accesso con cache = 100 / 125 × T medio istruzioni + 25 / 125 × T medio dati =

= 0,8 × 7,4 + 0,2 × 65 = 5,92 + 13 = 18,92 cicli di clock

Punto 4:

Rapporto di prestazioni dell'architettura senza cache rispetto a quella con cache =

= T medio di accesso senza cache / T medio di accesso con cache =

= 10 / 18,92 = 0,52 ≈ 0,5

Osservazione: il rapporto di prestazioni è assai minore di uno (di fatto vale circa un mezzo), pertanto l'architettura con cache si comporta decisamente peggio di quella senza cache; ciò è indesiderabile, ma se ne intuisce la ragione così: gli eventi di miss sono piuttosto frequenti, poiché il 25 % di istruzioni accede a dato e il 10 % di tali accessi incorre in miss, dunque il 2,5 % di istruzioni incorre in miss (come regola empirica una cache ben funzionante dovrebbe avere < 1 % di accessi che incorrono in miss); inoltre la penalità di miss è alquanto elevata, poiché la memoria centrale è 10 volte più lenta della memoria cache e il blocco da caricare da memoria centrale in memoria cache a seguito di miss, è di ben 64 parole ossia è piuttosto grosso; avendosi dunque miss frequenti e penalità di miss elevata, non è irragionevole che la memoria cache causi degrado di prestazione invece di aumento; si supponga che il miss rate di cache dati sia 1 % invece che 10 % ossia un ordine di grandezza inferiore, e si ricalcoli il rapporto di prestazione …

pipelining

Si consideri il frammento seguente di codice scritto in linguaggio macchina 68000:

CODICE SIMBOLICO			COMMENTO IN RTL
	ORG	...	// allinea indirizzo a ...
A:	**DC**	...	// definisce indirizzo A e inizializza cella di mem. A con ...
B:	**DC**	...	// definisce indirizzo B e inizializza cella di mem. B con ...
	MOVE	#20, D2	// D2 ← 20
	MOVE	A, D1	// D1 ← [A]
	ADD	D1, D2	// D2 ← [D1] + [D2]
	MOVE	D2, B	// B ← [D2]

Valgono le ipotesi seguenti:

- il processore ha una pipeline a quattro stadi: prelievo P; decodifica D; esecuzione E o accesso a dato in memoria M, rispettivamente; e scrittura S; il terzo stadio funziona come E o M secondo il caso
- ciascuno stadio P, D, E o M, e S lavora esattamente in un ciclo di clock
- ogni accesso a dato in memoria richiede un solo ciclo di clock, tranne quando c'è miss
- la MOVE che carica una costante in un registro ingombra due parole di memoria, mentre ogni altra MOVE – anche se ha un indirizzo come argomento – ingombra una sola parola di memoria
- il processore ha **due cache indipendenti**: una d'istruzione e una di dato, con due porte separate

Si tracci il diagramma di pipelining, inserendo gli stalli opportuni per risolvere le dipendenze di dato e **supponendo che il primo accesso a dato** (istruzione MOVE A, D1) **sia un miss** che richiede un totale di quattro cicli di clock per la soluzione, mentre il secondo (istruzione MOVE D2, B) è uno hit e richiede un solo ciclo di clock. Si usi la tabella seguente (il numero di colonne non è significativo).

---

	cicli di clock																			
ISTR.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
MOVE																				
MOVE																				
ADD																				
MOVE																				

soluzione

	cicli di clock																			
ISTR.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
MOVE	P	P	D	E	S															
MOVE			P	D	*stallo* M			M	S											
ADD				P	*stallo* D					D	E	S								
MOVE										P	*stallo* D		D	M	S					

Osservazione: la prima MOVE espleta la fase P in due cicli di clock poiché l'istruzione ingombra due parole; la fase P della seconda MOVE parte con ritardo di un ciclo per liberare lo stadio P; la seconda MOVE stalla la fase M per tre cicli poiché c'è miss con tempo totale (stallo ed esecuzione) di quattro cicli; la ADD stalla la fase D per cinque cicli in attesa del risultato di MOVE; la fase P della terza MOVE parte con ritardo di cinque cicli per liberare lo stadio P; e la terza MOVE stalla la fase D per due cicli in attesa del risultato di ADD.

spazio libero per brutta copia