Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria

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

AXO – Architettura dei Calcolatori e Sistemi Operativi

esame di giovedì 27 febbraio 2014

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 possibile lasciare l'aula conservando il tema della prova in corso.

Tempo a disposizione: 1h:30m (una parte) 3h:00m (completo).

	punteggio approssimativo	**I parte ☐**	**II parte ☐**	**completo ☐**
esercizio 1	**5**			
esercizio 2	**6**			
esercizio 3	**5**			
esercizio 4	**5**			
esercizio 5	**5**			
esercizio 6	**6**			
voto finale				

ATTENZIONE: alcuni esercizi sono suddivisi in parti.

esercizio n. 1 – thread e parallelismo

Si consideri il programma C seguente (gli "#include" sono omessi):

```
pthread_mutex_t OMEGA = PTHREAD_MUTEX_INITIALIZER;
sem_t ONE, TWO;
int global = 0;

void * A (void * arg) {
   int local = 0;
   sem_wait (&ONE);                                   /* istruzione A */
   pthread_mutex_lock (&OMEGA);
      sem_wait (&TWO);                                /* istruzione B */
   pthread_mutex_unlock (&OMEGA);
   return NULL;
} /* end A */

void * B (void * arg) {
   int local = 0;
   pthread_mutex_lock (&OMEGA);
      sem_post (&ONE);
      sem_post (&TWO);
   pthread_mutex_unlock (&OMEGA);                     /* istruzione C */
   return NULL;
} /* end B */

void * C (void * arg) {
   sem_wait (&TWO);
   sem_wait (&TWO);                                   /* istruzione D */
   global = 1;
   return NULL;
} /* end C */

void main ( ) {
   pthread_t TH_1, TH_2, TH_3;
   sem_init (&ONE, 0, 0);
   sem_init (&TWO, 0, 1);
   pthread_create (&TH_1, NULL, A, NULL);
   pthread_create (&TH_2, NULL, B, NULL);
   pthread_create (&TH_3, NULL, C, NULL);
   pthread_join (TH_2, NULL);
   pthread_join (TH_3, NULL);                         /* istruzione E */
   pthread_join (TH_1, NULL);
   return;
} /* end main */
```

Si completi la tabella qui sotto **indicando lo stato di esistenza della variabile locale** nell'istante di tempo specificato da ciascuna condizione, così: se la variabile **esiste**, si scriva ESISTE; se **non esiste**, si scriva NON ESISTE; e se può essere **esistente o inesistente**, si scriva PUÒ ESISTERE. Ogni casella della tabella va riempita in uno dei tre modi (non va lasciata vuota).

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede lo stato che la variabile o il parametro assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

condizione	**variabile locale**	
	local in **TH_1**	*local* in **TH_2**
subito dopo istr. **A**	*ESISTE*	*PUÒ ESISTERE*
subito dopo istr. **C**	*PUÒ ESISTERE*	*ESISTE*
subito dopo istr. **D**	*ESISTE*	*PUÒ ESISTERE*
subito dopo istr. **E**	*ESISTE*	*NON ESISTE*

Si completi la tabella qui sotto, **indicando i valori delle variabili globali** (sempre esistenti) nell'istante di tempo specificato da ciascuna condizione. Il **valore** della variabile va indicato così:

- intero, carattere, stringa, quando la variabile ha un valore definito; oppure X quando è indefinita
- se la variabile può avere due o più valori, li si riporti tutti quanti
- il semaforo può avere valore positivo o nullo (non valore negativo)

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede il valore (o i valori) che la variabile ha tra lo statement X e lo statement immediatamente successivo del thread indicato.

condizione	**variabile globale**		
	ONE	*TWO*	*global*
subito dopo istr. **A**	*0*	*0 / 1 / 2*	*0 / 1*
subito dopo istr. **B**	*0*	*0 / 1*	*0*
subito dopo istr. **C**	*0 / 1*	*0 / 1 / 2*	*0 / 1*
subito dopo istr. **D**	*0 / 1*	*0*	*0*

Il sistema va sempre in stallo (deadlock), in due casi diversi. Qui **si indichino** le primitive dove si bloccano i thread (uno solo o più di uno), precisando il valore (o i valori) della variabile `global`:

caso	**TH_1**	**TH_2**	**TH_3**	*global*
1	`wait TWO`	–	–	*0 / 1*
2	–	–	`seconda wait TWO`	*0*

esercizio n. 2 – processi e sistema operativo

prima parte – commutazione tra processi

Sono dati due **processi P** e **S**. Il processo **P** esegue il programma **CODICE_UNO** e crea il processo figlio **Q**. Il processo **S** (che non è figlio né di **P** né di **Q**) esegue il programma **CODICE_DUE**. Nel sistema non ci sono altri processi utente oltre a **P**, **S** e **Q**.

```
/* programma CODICE_UNO.c eseguito dai processi P e Q                */
main ( ) {
     char vet_P, vet_Q [...];    int fd1, fd2;    pid_t pid;
(1)  fd1 = open ("/acso/file1", O_RDWR);          /* 4 blocchi */
(2)  write (fd1, vet_P, 540);                     /* 2 blocchi */
     pid = fork ( );
     if (pid == 0) {        /* codice eseguito da Q figlio di P */
(3)     lseek (fd1, 48, 0); /* 0 si riferisce a inizio file     */
(4)     read (fd1, vet_Q, 1024);                  /* 4 blocchi */
(5)     fd2 = open ("/acso/file2", O_RDWR);       /* 4 blocchi */
(6)     write (fd2, vet_Q, 1024);                 /* 4 blocchi */
        ...
        exit (1);
     } else {               /* codice eseguito dal padre P      */
(7)     read (fd1, vet_P, 10);                    /* 1 blocco  */
        pid = wait (&status);
        ...
        exit (0);
    } /* if */
} /* CODICE_UNO */
```

```
/* programma CODICE_DUE.c eseguito dal processo S                    */
main ( ) {
     char vet_S [...];    int fd;
(8)  fd = open ("/acso/file2", O_RDWR);           /* 4 blocchi */
(9)  lseek (fd, 560, 1);   /* 1 si riferisce a posizione corr. */
(10) write (fd, vet_S, 512);                      /* 1 blocco  */
     ...
     exit (2);
} /* CODICE_DUE */
```

Ulteriori specifiche:

1. i processi utente hanno una priorità associata, il processo "idle" ha priorità minima e nel sistema non ci sono altri processi oltre a quelli utente e a "idle"
2. quando è necessario, sono indicate le priorità dei processi attivi nel sistema
3. quando un processo diventa pronto e ha priorità maggiore di quello in esecuzione, va attivato lo scheduler
4. il buffer del driver di standard output ha dimensione di **50 caratteri**
5. per le operazioni sui file che – nell'esercizio considerato – implicano trasferimento di blocchi:
    a. l'interruzione di fine DMA è associata al trasferimento di un singolo blocco di file: evento *DMA_in* per lettura di un blocco da file ed evento *DMA_out* per scrittura di un blocco su file
    b. il numero di blocchi da trasferire è indicato esplicitamente come commento nel codice
6. le chiamate di sistema "wait" e "waitpid" invocano la funzione "Sleep_on" su un evento opportuno
7. per completare la tabella delle commutazioni, si faccia riferimento alla notazione vista a lezione

Si completino le parti mancanti della tabella di commutazione dei processi. Nella tabella sono previste righe da completare, dove:

a. è specificato l'evento (con informazioni aggiuntive); qui sono da completare le parti relative allo stato dei processi e – se richiesto – ai moduli del S.O. con il contesto

b. è specificato lo stato dei processi raggiunto dopo il verificarsi dell'evento; qui sono da completare i campi relativi all'evento (con eventuali informazioni aggiuntive) e – se richiesto – ai moduli del S.O. con il contesto

   NOTA BENE:

   - l'evento è sempre determinabile univocamente dallo stato raggiunto, dall'evoluzione precedente dei processi, dal codice dei programmi e dalle ulteriori specifiche di sistema
   - se l'evento è un interrupt, va usata la notazione "*n* interrupt", indicando esattamente il numero di interruzioni che si sono verificate

evento (associato al nome del processo dove si verifica)	**informazioni aggiuntive**	**moduli del SO invocati per gestire l'evento**	**processo dov'è gestito il modulo**	**stato dei processi dopo la gestione dell'evento**		
				P	**Q**	**S**
				pronto per scadenza del quanto di tempo	**non esiste**	**esecuzione**
S: lseek ( )		*G_SVC_1* *lseek* *G_SVC_2/3*	*S* *S* *S*	*pronto*	*non esiste*	*esec*
S: interrupt da Real Time Clock (orologio)	il quanto di tempo di S è scaduto P < S	*R_int (CK)* *Preempt* *R_int (CK)*	*S* *S* *S*	*pronto*	*non esiste*	*esec*
S: write	*write ha inizializzato il DMA in scrittura*	*G_SVC_1* *write* *Sleep_on_1 (E1)* *Change* *Preempt_2* *R_int (CK)*	*S* *S* *S* *S – P* *P* *P*	esec	non esiste	attesa(E1)
P: pid = fork		*G_SVC_1* *fork* *G_SVC_2/3*	*P* *P* *P*	esec	pronto	attesa(E1)
P: read	read ha inizializzato il DMA in lettura	*G_SVC_1* *read* *Sleep_on_1 (E2)* *Change* *G_SVC_2/3*	*P* *P* *P* *P – Q* *Q* *Q*	*attesa (E2)*	*esec*	*attesa (E1)*
Q: fd2 = open	open ha inizializzato il DMA in lettura	*G_SVC_1* *open* *Sleep_on_1 (E3)* *Change*	*Q* *Q* *Q* *Q – idle*	*attesa (E2)*	*attesa (E3)*	*attesa (E1)*
idle: 3 interrupt da DMA	2 da DMA_in per Q 1 da DMA_out per S riportare i moduli del SO solo per l'ultimo interrupt	*R_int (DMA_out)* *Wake_up (E1)* *Preempt_1* *Change* *Sleep_on (E1)* *write* *G_SVC_2/3*	*idle* *idle* *idle* *idle* *idle – S* *S* *S*	*attesa (E2)*	*attesa (E3)*	*esec*

TABELLA DI COMMUTAZIONE DEI PROCESSI *(fine)*

seconda parte – funzioni del file system
Si consideri nuovamente il codice dei programmi dati nella prima parte e in particolare le chiamate di sistema identificate dai numeri d'ordine da **1** a **10**.

Relativamente a questa seconda parte valgono le specifiche seguenti:

- l'area buffer in memoria centrale gestita dal SO per le operazioni su file è costituita da **4 buffer** di dimensione opportuna che possono essere assegnati indifferentemente ai blocchi di un qualsiasi file o catalogo del sistema; in caso di mancanza di spazio l'area buffer viene gestita con **politica LRU**
- per **tutte le operazioni** su file
  (a) la dimensione del blocco trasferito da o verso file tramite **DMA** è **512 byte**
  (b) l'interruzione di fine DMA è associata al trasferimento di un singolo blocco
  (c) le chiamate di sistema sono dipendenti e pertanto i blocchi allocati/letti dalle precedenti chiamate di sistema rimangono a disposizione in memoria finché possibile
  (d) per poter scrivere su file è necessario avere caricato i blocchi in memoria
- per l'**apertura** del file è **sempre** necessario accedere a:
  (a) un blocco per l'accesso allo *I-node* di ogni cartella (catalogo) o file presente nel nome-percorso (pathname); **una volta letti da disco, gli *I-node* dei file e delle cartelle vengono scritti e mantenuti sempre in memoria in lista apposita non facente parte dell'area buffer**, fino a quando il file/cartella è referenziato da almeno un processo
  (b) un blocco per il contenuto di ogni cartella (catalogo) presente nel nome-percorso (pathname)

Si consideri la seguente sequenza di esecuzione delle chiamate di sistema: **2 3 4 7 5 6**

Per ciascuna della chiamate, **si indichino** nella tabella predisposta (a pagina seguente):

- il numero totale di interruzioni di fine DMA che si verificano affinché l'operazione possa completarsi
- la sequenza di accessi agli I-node (notazione "I-node [X]", con X = numero i-node) e ai blocchi (con la notazione "blocco Y") specificando se a memoria (M) o a disco (D) e per i blocchi se in lettura o scrittura
- lo stato dei 4 buffer alla fine della gestione di ogni chiamata riportando per ciascun buffer il blocco presente e il file/cartella a cui appartiene, oppure se è libero (L)

N.B.: La prima riga della tabella è già compilata.

---

contenuto del volume

I-lista: ⟨ 0, dir, 4 ⟩ ⟨ 6, dir, 20 ⟩ ⟨ 32, norm, (600, 601, 602, 603, 604, 605, 606) ⟩
⟨ 48, norm, (800, 812, 700, 703, 705) ⟩

blocco 4: ... ⟨ 6, acso ⟩ ...

blocco 20: ... ⟨ 32, file1 ⟩ ⟨ 48, file2 ⟩ ...

tabella da completare

chiamata di sistema	n. interrupt di fine DMA	sequenza di accessi in memoria o su disco	stato dei buffer
(2) **write** (fd1, vet_P, 540)	0 1 0 1 0	I-node [32] M read blocco 600 D write blocco 600 M read blocco 601 D write blocco 601 M	BUF_1 = 4 root BUF_2 = 20 acso BUF_3 = 600 file1 BUF_4 = 601 file1
(3) **lseek** (fd1, 48, 0)	*0*	*nessuno*	BUF_1 = *4 root* BUF_2 = *20 acso* BUF_3 = *600 file1* BUF_4 = *601 file1*
(4) **read** (fd1, vet_Q, 1024)	*0* *0* *0* *1*	*I-node [32] M* *read blocco 600 M* *read blocco 601 M* *read blocco 602 D*	BUF_1 = *602 file1* BUF_2 = *20 acso* BUF_3 = *600 file1* BUF_4 = *601 file1*
(7) **read** (fd1, vet_P, 10)	*0*	*I-node [32] M* *read blocco 602 M*	BUF_1 = *602 file1* BUF_2 = *20 acso* BUF_3 = *600 file1* BUF_4 = *601 file1*
(5) fd2 = **open** ("/**acso/file2**", O_RDWR)	*0* *1* *0* *0* *0* *1*	*I-node [0] M* *write blocco 600 D* *read blocco 4 M* *I-node [6] M* *read blocco 20 M* *I-node [48] D*	BUF_1 = *602 file1* BUF_2 = *20 acso* BUF_3 = *4 root* BUF_4 = *601 file1*
(6) **write** (fd2, vet_Q, 1024)	*0* *1* *1* *0* *1* *0*	*I-node [48] M* *write blocco 601 D* *read blocco 800 D* *write blocco 800 M* *read blocco 812 D* *write blocco 812 M*	BUF_1 = *812 file2* BUF_2 = *20 acso* BUF_3 = *4 root* BUF_4 = *800 file2*

esercizio n. 3 – memoria virtuale

Un sistema dotato di memoria virtuale con paginazione e segmentazione di tipo UNIX, è caratterizzato dai parametri seguenti: la memoria centrale fisica ha capacità di **16 K byte**, quella logica di **16 K byte** la pagina è di **2048** byte. Pertanto un **indirizzo virtuale** è codificato con **14 bit**, dei quali gli **11 bit** meno significativi rappresentano lo **spiazzamento** (*offset*) all'interno della pagina.

indir. virtuale	prog. X	prog. Y
0	*CX0*	*CY0*
1	*CX1*	*CY1*
2	*CX2*	*DY0*
3	*DX0*	*DY1*
4		
5	*COND*	*COND*
6		
7	*PX0*	*PY0*

Si chiede di svolgere i punti seguenti:

a. Nel sistema vengono creati tre processi, indicati nel seguito con **P**, **Q** e **R**. I programmi eseguiti da tali processi sono due: **X** e **Y**. La dimensione iniziale dei segmenti dei programmi è la seguente:

CX: **6 K** DX: **2 K** PX: **2 K** COND: **2 K**

CY: **4 K** DY: **4 K** PY: **2 K** COND: **2 K**

La pagina condivisa è allocata lasciando libera una pagina dopo il segmento dati.

Nella tabella qui accanto **si inserisca** la struttura in pagine della memoria virtuale (mediante la notazione CX0 CX1 DX0 PX0 … CY0 …).

b. In un certo istante di tempo $\mathbf{t_0}$ sono terminati, nell'ordine, gli eventi seguenti:

1. creazione del processo **P** e lancio del programma **X** (*fork* di P ed *exec* di X) *alloca CP1/1 e PP0/7*
2. **P** accede alla variabile all'indirizzo virtuale assoluto **1EFF hex**, corrispondente alla pagina virtuale **3** *alloca DP0/3*
3. **P** crea il processo figlio **Q** (P esegue *fork*) *condividi CQ1 = CP1, DQ0 = DP0, alloca PQ0*
4. **Q** alloca la pagina condivisa *dealloca DQ0; alloca COND*
5. **P** esegue *exit* *dealloca CP1, DP0, PP0*
6. **Q** esegue *fork* e crea il processo figlio **R** *condividi CR1 = CQ1, COND, e alloca PR0*

Valgono le convenzioni seguenti:

- un programma viene lanciato caricando **soltanto** la pagina di codice con l'istruzione di partenza e una pagina di pila, in quest' ordine
- il caricamento di pagine ulteriori avviene in **demand paging** (ossia su richiesta)
- l'indirizzo dell'istruzione di partenza di **X** è **08A0 hex** (virtuale assoluto), corrispondente alla pagina virtuale **1**
- l'indirizzo dell'istruzione di partenza di **Y** è **0F0E hex** (virtuale assoluto), corrispondente alla pagina virtuale **1**
- il numero ***R*** di pagine residenti vale **3**
- viene utilizzato l'algoritmo **LRU**
- l'allocazione delle pagine virtuali in pagine fisiche, avviene sempre in **sequenza**

ATTENZIONE In una *fork* l'allocazione delle pagine del figlio avviene in sequenza di pagina virtuale: codice, eventuali dati e segmento dati condiviso, e pila.

Si completi la **tab. 1A** (parte sinistra) con l'allocazione fisica delle pagine dei tre processi all'istante $\mathbf{t_0}$, e si completi la **tab. 1B** con il contenuto della MMU, ipotizzando che le righe della tabella siano state allocate ordinatamente man mano che venivano allocate le pagine di memoria virtuale.

memoria fisica	
indir. fisico	**pagine allocate al tempo t_0**
0	*(CP1) = CQ1 = CR1*
1	*(PP0) PR0*
2	*(DP0) = (DQ0)*
3	*PQ0*
4	*COND*
5	
6	
7	

tab. 1A

MMU			
proc.	**NPV**	**NPF**	**valid bit**
(P) R	*(CP1 / 1) CR1 / 1*	*(0) 0*	*1 0 1*
(P) R	*(PP0 / 7) COND / 5*	*(1) 4*	*1 0 1*
(P) R	*(DP0 / 3) PR0 / 7*	*(2) 1*	*1 0 1*
Q	*CQ1 / 1*	*0*	*1*
(Q) Q	*(DQ0 / 3) COND / 5*	*(2) 4*	*1 0 1*
Q	*PQ0 / 7*	*3*	*1*

tab. 1B

c. In un certo istante di tempo $t_1 > t_0$ sono terminati, nell'ordine, gli eventi seguenti:
   7. **R** chiama *exec* e passa a eseguire il programma **Y** *dealloca CR1, COND, PR0; alloca CR1, PR0*
   8. **Q** chiama la funzione all'indirizzo virtuale assoluto **0C88** hex *accede alle pagine CQ1, PQ0*
   9. **R** esegue un accesso dati all'indirizzo virtuale assoluto **1C64** hex *alloca e accede a DR1*

   Si completi la **tab. 2A** supponendo che, se occorre una pagina fisica libera, sia sempre usata la pagina fisica libera con indirizzo minore, e si completi la **tab. 2B** con il contenuto della MMU all'istante t_1.

memoria fisica	
indir. fisico	**pagine allocate al tempo t_1**
0	*CQ1 = (CR1)*
1	*(PR0) CR1*
2	*PR0*
3	*PQ0*
4	*COND*
5	*DR1*
6	
7	

tab. 2A

MMU			
proc.	**NPV**	**NPF**	**valid bit**
(R) R	*(CR1 / 1) CR1 / 1*	*(0) 1*	*1 0 1*
(R) R	*(COND / 5) PR0 / 7*	*(4) 2*	*1 0 1*
(R) R	*(PR0 / 7) DR1 / 3*	*(1) 5*	*1 0 1*
Q	*CQ1 / 1*	*0*	*1*
Q	*COND / 5*	*4*	*1*
Q	*PQ0 / 7*	*3*	*1*

tab. 2B

esercizio n. 4 – logica digitale

prima parte – logica combinatoria

Data la funzione combinatoria F seguente:

$$F(a, b, c, d) = !(a+b+c+d) + (!ac+a!c)!b!d + (!b!d+bd)!ac + abcd$$

a) La **si trasformi** in somma di prodotti (SOP) senza curare la minimizzazione:

F (*a*, *b*, *c*, *d*) = *!a !b !c !d + !a !b c !d + a !b !c !d + !a !b c !d + !a b c d + a b c d* _____

*(trasf. De Morgan del primo termine e calcolo dei prodotti del secondo e del terzo)*_____

b) Poi **si disegni** la mappa di Karnaugh della funzione F, se ne mettano in evidenza gli implicanti primi, e per ogni implicante primo individuato si scriva la forma algebrica corrispondente:

riportando sulla mappa gli implicanti trovati (il secondo e il quarto sono identici), si vede che essi sono raggruppabili in modo da coprirli con tre implicanti primi essenziali:

(0, 2) = !a !b !d (0, 8) = !b !c !d (7, 15) = b c d

a b \ c d	*00*	*01*	*11*	*10*
00	*1*			*1*
01			*1*	
11			*1*	
10	*1*			

c) Poi **si minimizzi** la funzione F tramite il metodo delle mappe di Karnaugh e **si scriva** la forma minima ottenuta. Qualora esistano più forme minime, le si indichino tutte (il numero di righe non è significativo):

F (*a*, *b*, *c*, *d*) = *!a !b !d + !b !c !d + b c d* *(i tre implicanti, tutti essenziali)*_____

F (*a*, *b*, *c*, *d*) = _____

d) **Si disegni** una rete combinatoria che realizza la funzione F sintetizzata al punto precedente, utilizzando solamente porte a **due** ingressi (e negatori): *banale, lasciato al lettore*

e) **Si calcoli** il ritardo massimo (percorsi critici) della rete disegnata prima, supponendo che la porta NOT abbia ritardo di **1 ns,** la porta AND a due ingressi abbia ritardo di **2 ns** e la porta OR a due ingressi abbia ritardo di **3 ns** (non si consideri il ritardo di propagazione lungo i conduttori):

ritardo (F) = *1 (NOT) + 2 (AND in cascata) × 2 ns + 2 (OR in cascata) × 3 ns = 11 ns*_____

seconda parte – logica sequenziale

Sia dato il circuito sequenziale **con un ingresso I e due uscite Z1 e Z2** descritto dalle equazioni logiche seguenti:

$$D = I \ \textbf{xor} \ Q$$

$$Z1 = Q$$

$$Z2 = I \ \textbf{xor} \ (\textbf{not} \ Q)$$

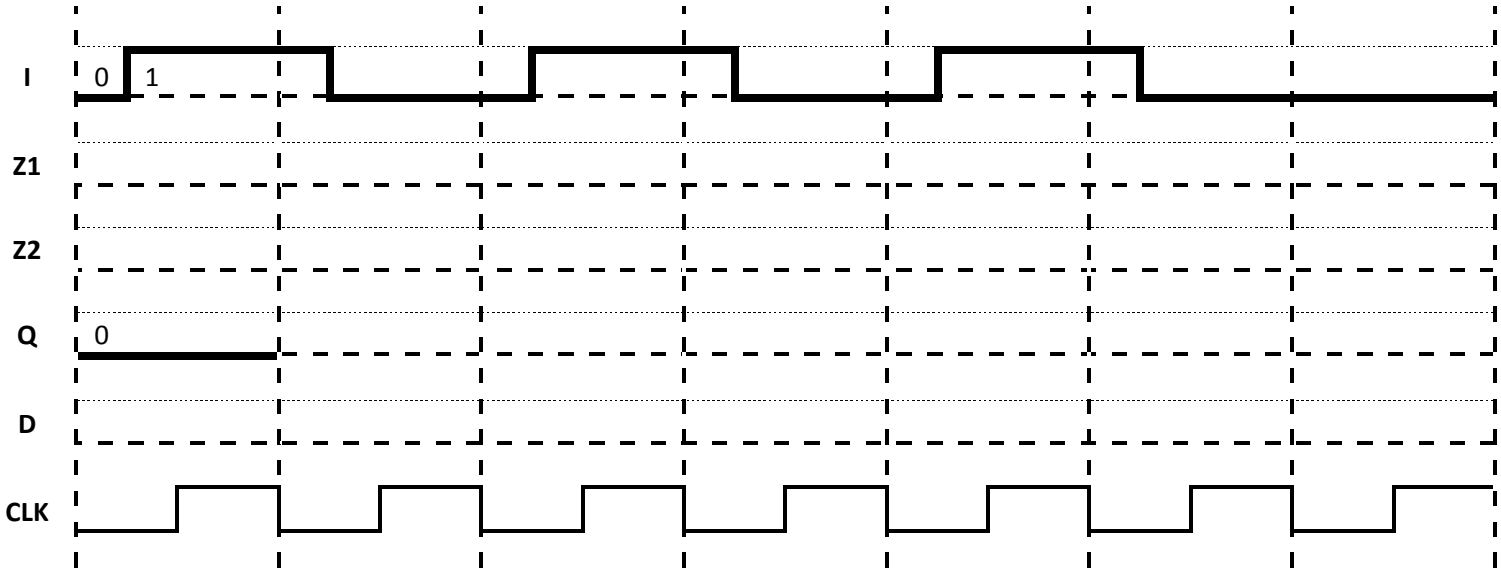
Il circuito comprende **un** bistabile master / slave di tipo D (D, Q), con D ingresso del bistabile e Q stato / uscita del bistabile.

Si chiede di completare il diagramma temporale riportato qui sotto. Si noti che:

- si devono trascurare completamente i ritardi di propagazione delle porte logiche NOT e XOR, e i ritardi di campionamento e commutazione del bistabile
- il bistabile impiegato campiona e memorizza l'ingresso sul **fronte di salita** del clock e sul **fronte di discesa** del clock mostra il valore memorizzato; durante i due livelli del clock è insensibile agli ingressi e mantiene fissa l'uscita

---

diagramma temporale da completare



soluzione

D è uguale all'ingresso I quando Q = 0, è la negazione dell'ingresso I quando Q = 1

Q è costante per tutto un intervallo, e ha il valore che D aveva in corrispondenza del precedente fronte di salita del clock

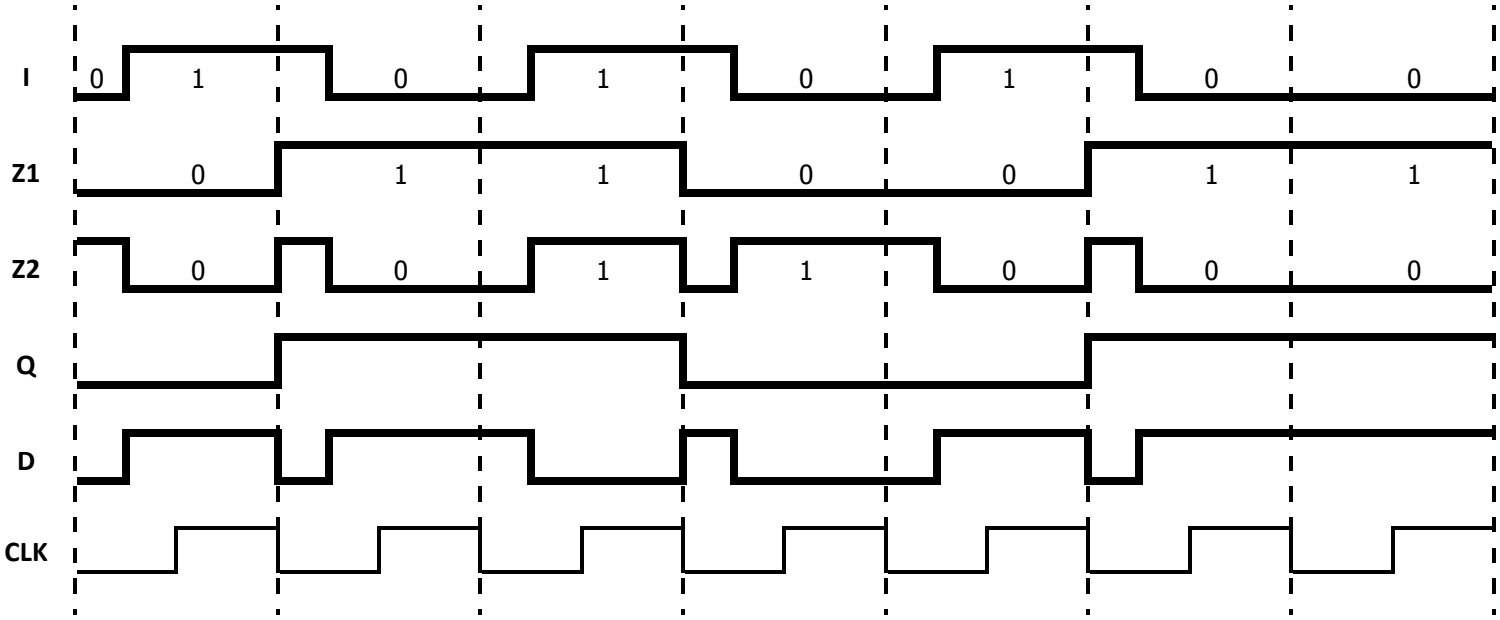
Z1 dipende solo dallo stato (è uguale a Q) quindi deve rimanere costante nell'intervallo

Z2 dipende da stato e ingresso, quindi può variare al variare dell'ingresso

Procedimento di calcolo dei valori in un intervallo:

- *si pone Q uguale a D al fronte di salita a metà dell'intervallo precedente*
- *si pone Z1 uguale a Q*
- *si calcola D nell'intervallo, istante per istante*
- *si calcola Z2 nell'intervallo, istante per istante*

Poi si passa all'intervallo successivo. Nel primo intervallo Q è ovviamente già dato (è lo stato iniziale, non calcolabile perché non c'è l'intervallo precedente).



Descrizione del comportamento ai morsetti:

Z1 è il bit di parità di tutti i bit di ingresso arrivati in precedenza (cioè non considerando quello attualmente in arrivo): si vedano le sequenze di 0 e di 1 riportate nel diagramma (per I si devono considerare i valori in corrispondenza dei fronti di salita del clock). Il primo valore è 0 perché in precedenza non è arrivato nulla: niente 1 cioè zero 1 quindi parità pari.

Z2 è uguale all'ingresso quando Z1 = 1, è la negazione dell'ingresso quando Z1 = 0. Se guardassimo solo i valori in corrispondenza del fronte di salita del clock, troveremmo la sequenza di ingresso, con negati i bit in corrispondenza degli zeri della sequenza Z1 dei bit di parità.

esercizio n. 5 – linguaggio macchina

Si chiede di tradurre in linguaggio macchina simbolico (linguaggio assemblatore) 68000 la funzione *funz* riportata sotto. Nel tradurre non si tenti di accorpare od ottimizzare insieme istruzioni C indipendenti. La memoria ha parole da **32 bit**, indirizzi da **32 bit** ed è indirizzabile per **byte**. Le variabili intere 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 rispettivi chiamanti 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

La funzione *g* utilizzata da *funz*, restituisce un intero e ha la testata seguente:

```
int g (int, int)
```

Si chiede di:

1. **Scrivere** (in tabella 1) la traduzione delle definizioni delle variabili globali (utilizzando quindi solamente direttive all'assemblatore).
2. **Disegnare** l'albero sintattico dell'espressione aritmetica presente nella parte sinistra della condizione dello statement IF della funzione *funz*, ossia l'albero di `(r * g (r + c, RIGHE))`.
3. **Disegnare** l'albero sintattico dell'espressione necessaria per valutare la parte destra della condizione dello statement IF della funzione *funz*, ossia l'albero di `MATRICE[r][c]`, dove *r* è l'indice di riga e *c* l'indice di colonna (attenzione: questo albero sintattico deve contenere l'espansione di tutte le operazioni necessarie per trovare l'indirizzo dell'elemento e leggerlo). Si ricorda che in linguaggio C le matrici sono allocate in memoria per righe, cioè è allocata tutta la prima riga, poi tutta la seconda, ecc.
4. **Scrivere** (in tabella 2) il codice in linguaggio assemblatore 68000 della funzione *funz*, coerentemente con le risposte ai punti precedenti e indicando gli identificativi numerici degli operatori dell'albero sintattico nel codice che traduce l'espressione. Nel codice prodotto non devono essere presenti valori numerici (eccetto ovviamente quelli posti nelle direttive EQU).

```
// costanti e variabili globali da allocare staticamente

#define DIM_ELM  4    // dimensione di un intero in byte
#define RIGHE    5
#define COLONNE  4
#define UGUALI   0
#define DIVERSI  1

int MATRICE [RIGHE] [COLONNE];      // matrice di interi

// funzione funz
int funz (int r, int c) {
    if ((r * g (RIGHE, r + c) == MATRICE [r] [c]) {
       return UGUALI;
    } else {
       return DIVERSI;
    } /* if */
} /* funz */
```

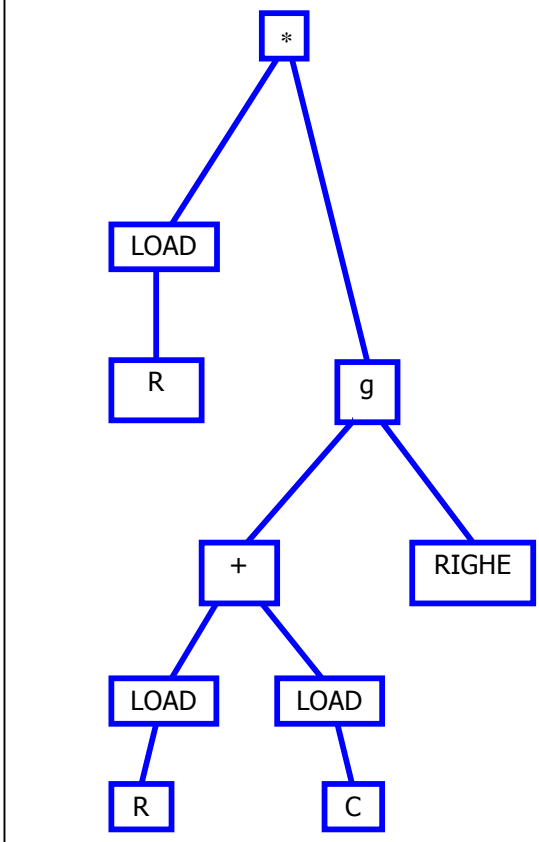
Tabella 1: TRADUZIONE DELLE COSTANTI E VARIABILI GLOBALI

DIM_ELM:	***EQU***	*4*	*// dimensione dell'elemento della matrice*
RIGHE:	***EQU***	*5*	*// numero di righe della matrice*
COLONNE:	***EQU***	*4*	*// numero di colonne della matrice*
UGUALI:	***EQU***	*0*	*// esito affermativo*
DIVERSI:	***EQU***	*1*	*// esito negativo*
MATRICE:	***DS.L***	*20*	*// varglob matrice con RIGHE x COLONNE*

DISEGNARE QUI GLI ALBERI SINTATTICI

Per evitare confusione, si usi l'operatore **DEREF** per indicare l'accesso a una cella di memoria in base a un indirizzo, e l'operatore **LOADADDR** per indicare il caricamento di un indirizzo in un registro.

(r * g (RIGHE, r + c))



MATRICE [r] [c]

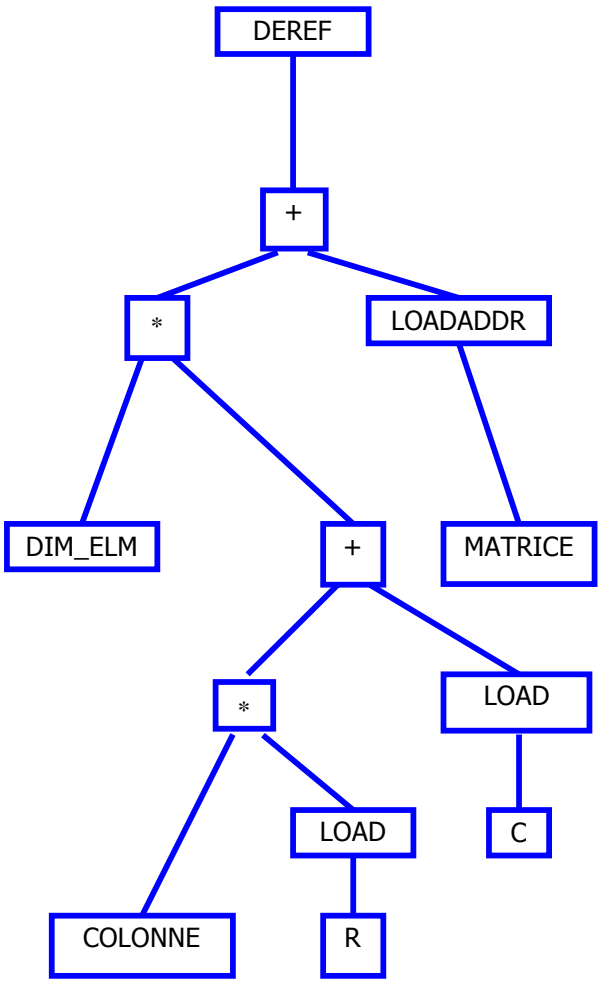


Tabella 2: CODICE MACCHINA DI FUNZ

FUNZ:	**LINK**	FP, *#0*	*// collega*
R:	**EQU**	*8*	*// spiazzamento di varloc r*
C:	**EQU**	*12*	*// spiazzamento di varloc c*
	MOVEM.L	*D0-D5/A0, -(SP)*	*// salva registri*
	MOVE.L	*R(FP), D0*	*// carica r*
	MOVE.L	*C(FP), D1*	*// carica c*
	ADD.L	*D0, D1*	*// calcola r + c*
	MOVE.L	*D1, -(SP)*	*// impila 1° param di g*
	MOVE.L	*#RIGHE, -(SP)*	*// impila 2° param di g*
	BSR	*G*	*// chiama funz g*
	ADDA.L	*#4, SP*	*// abbandona 2° param di g*
	MOVE.L	*(SP)+, D2*	*// spila valusc di g*
	MOVE.L	*R(FP), D3*	*// carica r*
	MULS	*#COLONNE, D3*	*// calcola COLONNE * r*
	MOVE.L	*C(FP), D4*	*// carica c*
	ADD.L	*D3, D4*	*// calcola (COLONNE * r) + c*
	MULS	*#DIM_ELM, D4*	*// calcola DIM_ELM * ((COLONNE * r) + c)*
	MOVEA.L	*#MATRICE.L, A0*	*// carica ind. iniz. di MATRICE*
	ADDA.L	*D4, A0*	*// calcola ind. di elem. MATRICE [r][c]*
	MOVE.L	*(A0), D5*	*// carica risultato di espr. 2*
	CMP.L	*D2, D5*	*// confronta espressioni 1 e 2*
	BNE	*ELSE*	*// se condizione falsa vai a ELSE*
THEN:	***MOVE.L***	*#UGUALI, C(FP)*	*// sovrascrivi valusc*
	BRA	*ENDIF*	*// vai a ENDIF*
ELSE:	***MOVE.L***	*#DIVERSI, C(FP)*	*// sovrascrivi valusc*
ENDIF:	***MOVEM.L***	*(SP)+, D0-D5/A0*	*// ripristina registri*
	UNLK	*FP*	*// scollega*
	RTS		*// rientra*

esercizio n. 6 – memoria cache e perstazioni

Si consideri un sistema di memoria costituito da una memoria centrale di **16 M parole** e da una cache set-associativa (associativa a gruppi) a **4 vie** di **8 K parole** con blocchi da **32 parole**, che utilizza un algoritmo di sostituzione del blocco di tipo **LRU**.

La cache è inizialmente vuota. La CPU esegue un ciclo che preleva sequenzialmente dalla memoria un array di **8352** elementi di una parola, corrispondenti a **261 blocchi**, partendo dall'indirizzo **0**. Il ciclo viene eseguito per **5** volte.

Si risponda alle domande seguenti.

1) **Si mostri** la struttura dell'indirizzo di memoria (nella tabellla predisposta).

etichetta	indice	spiazz.
13 bit	*6 bit*	*5 bit*

La memoria ha 16 M parole: 24 bit per l'indirizzo.

I blocchi sono da 32 parole: 5 bit per identificare la parola nel blocco.

Nella cache (8 KB) ci sono: $2^{13} / 2^5 = 2^8 = 256$ blocchi. La memoria cache è set-associativa a quattro vie, dunque avrà $2^8 / 2^2 = 2^6 = 64$ insiemi di quattro blocchi. Sono pertanto necessari 6 bit per individuare l'insieme. L'etichetta è dunque di 13 bit.

2) **Si completi** la tabella **A** indicando il numero decimale dei blocchi dell'array che appartengono agli insiemi indicati (parte della prima riga è già compilata).

Tabella A

insieme (gruppo)	blocchi
0	0, 64*, 128, 196, 256*
1	*1, 65, 129, 197, 257*
2	*2, 66, 130, 198, 258*
3	*3, 67, 131, 199, 259*
4	*4, 68, 132, 200, 260*

Tutti i blocchi che in memoria hanno come valore 000000 per i bit di insieme, quindi i blocchi 0 (in binario 000 000000), 64 (001 000000), 128 (010 000000), 196 (011 000000) e 256 (100 000000) finiranno nell'insieme 000000, tutti i blocchi che hanno indirizzo terminante con 00001 (1, 65, 129, 197, 257) finiranno nell'insieme 000001, tutti i blocchi che terminano con 00010 (2, 66, 130, 198, 258) finiranno nell'insieme 000010, e così via.

3) **Si indichi** il numero di MISS che si verificano durante la prima esecuzione del ciclo (nella tabella predisposta), spiegano a lato sinteticamente il motivo.

n. di miss nella prima iterazione del ciclo
261 miss

Nel primo ciclo si verificano 261 miss, tanti quanti i blocchi del vettore, poiché inizialmente la memoria cache è vuota.

4) **Si indichi** in tabella **B** il numero decimale dei blocchi contenuti negli insiemi indicati al termine della prima iterazione del ciclo, la quale legge **8352** elementi.

Tabella B

insieme (gruppo)	blocchi
0	*64, 128, 196, 256*
1	*65, 129, 197, 257*
…	
4	*68, 132, 200, 260*
5	*5, 69, 133, 201*
6	*6, 70, 134, 202*
…	
63	*63, 127, 195, 255*

La cache si riempie con il blocco 255 che apparterrà all'insieme 63 (111111).

Quando bisogna caricare il blocco numero 256 si osserva che ha i bit di insieme pari a 000000, dunque il suo "posto" è nell'insieme 000000: tutto l'insieme è occupato e non c'è il blocco 256, dunque si verifica un miss e con l'algoritmo LRU verrà rimosso il blocco 0.

In modo analogo il blocco 257 (1 000 001) finirà nell'insieme 00001 dove verrà rimosso il blocco 1 e così fino al blocco 260 che finirà nell'insieme 3 sostituendo il blocco 3.

5) **Si calcoli** il numero di MISS che si verificano durante **ciascuna iterazione del ciclo dopo la prima** (esso è lo stesso per tutte e quattro le iterazioni e basta darlo per una iterazione). Si scriva il risultato nella tabella predisposta (e a lato si illustri il calcolo).

n. di miss in ciascuna iterazione del ciclo dopo la prima
25 miss per ciascuna

Nella seconda iterazione, quando serve il blocco 0 si verifica un miss e nella cache viene rimosso il blocco 64; quando serve il blocco 1, si verifica un miss e viene rimosso il blocco 65; quando serve il blocco 2, si verifica un miss e viene rimosso il blocco 66; così come avviene per il blocco 3 che sostituisce il blocco 67, e il blocco 4 che sostituisce il blocco 68.

Invece il blocco 5 e i successivi sono già in memoria. Avremo hit fino al blocco 63, quando di verificano 5 miss di seguito e andranno a sostituire i blocchi 128, 129, 130, 131, 132; e così via, hit fino quando si arriva al blocco 127 e poi 5 miss, idem fino al blocco 195 e poi 5 miss, e idem fino al blocco 255 e poi 5 miss, che vanno a sostituire i blocchi 0, …, 4 con 256, …, 261. In totale si verificano dunque 25 miss nella seconda iterazione del ciclo.

Si vede che la situazione si riproduce nella terza iterazione, nella quarta e nella quinta. Pertanto si verificano in tutto 25 miss per ciascuna iterazione del ciclo, a partire dalla seconda (compresa) fino alla quinta (compresa).

6) **Si calcoli** il fattore di miglioramento risultante dall'utilizzo della cache facendo l'ipotesi che il tempo di accesso alla memoria cache sia di **1 ns**, il tempo di accesso alla memoria centrale sia di **10 ns**, e che il bus dati abbia la dimensione della parola. Si scriva il risultato nella tabella predisposta (e a lato si illustri il calcolo).

fattore di miglioramento
3,1 volte

Tempo_senza_cache = 5 x 8352 elementi x 10 ns = 417.600 ns

Tempo_con_cache = 261 blocchi x 32 elementi_nel_blocco x (10 + 1) ns + 4 iterazioni del ciclo x (25 blocchi_miss x 32 elementi_nel_blocco x (10 + 1) ns + 236 blocchi_hit x 32 elementi_nel_blocco x 1 ns) = 91872 + 35200 + 7552 = 134.624 ns

Fattore di miglioramento = tempo senza cache / tempo con cache = 3,1 volte

spazio libero per brutta o continuazione

spazio libero per brutta o continuazione